HIPPODROME: Data Race Repair using Static Analysis Summaries

ANDREEA COSTEA, National University of Singapore, Singapore
ABHISHEK TIWARI, National University of Singapore, Singapore
SIGMUND CHIANASTA, National University of Singapore, Singapore
KISHORE R, National University of Singapore, Singapore
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore
ILYA SERGEY, National University of Singapore, Singapore

Implementing bug-free concurrent programs is a challenging task in modern software development. State-of-the-art static analyses find hundreds of concurrency bugs in production code, scaling to large codebases. Yet, fixing these bugs in constantly changing codebases represents a daunting effort for programmers, particularly because a fix in the concurrent code can introduce other bugs in a subtle way.

In this work, we show how to harness compositional static analysis for concurrency bug detection, to enable a new Automated Program Repair (APR) technique for data races in large concurrent Java codebases. The key innovation of our work is an algorithm that translates procedure summaries inferred by the analysis tool for the purpose of bug reporting into small local patches that fix concurrency bugs (without introducing new ones). This synergy makes it possible to extend the virtues of compositional static concurrency analysis to APR, making our approach *effective* (it can detect and fix many more bugs than existing tools for data race repair), *scalable* (it takes seconds to analyse and suggest fixes for sizeable codebases), and *usable* (generally, it does not require annotations from the users and can perform *continuous* automated repair). Our study conducted on popular open-source projects has confirmed that our tool automatically produces concurrency fixes similar to those proposed by the developers in the past.

CCS Concepts: • **Software and its engineering** → *Software defect analysis*; *Software maintenance tools*; • **Computing methodologies** → *Concurrent programming languages*.

Additional Key Words and Phrases: Concurrency, Program Repair, Static Analysis

1 INTRODUCTION

It is well acknowledged that implementing both correct and efficient concurrent programs is difficult [26]. While programmers have a robust understanding of sequential programs, their understanding of concurrently interacting processes is often incomplete, which may lead to subtle bugs. Once introduced, these bugs are hard to identify due to the inherently non-deterministic nature of concurrent executions. In other words, these issues can only be detected under selective thread schedulings which are challenging to reproduce during debugging. A number of tools have been

Authors' addresses: Andreea Costea, National University of Singapore, Singapore, Singapore, andreeac@comp.nus.edu.sg; Abhishek Tiwari, National University of Singapore, Singapore, Singapore, tiwari@comp.nus.edu.sg; Sigmund Chianasta, National University of Singapore, Singapore, Singapore, sigmund@u.nus.edu; Kishore R, National University of Singapore, Singapore, Singapore, kishore_r@u.nus.edu; Abhik Roychoudhury, National University of Singapore, Singapore, Singapore, abhik@comp.nus.edu.sg; Ilya Sergey, National University of Singapore, Singapore, Singapore, ilya@nus.edu.sg.

Manuscript submitted to ACM

introduced to (semi-) automatically detect concurrency bugs in real-world programs, thus facilitating their discovery and reproducibility [10, 18–20, 24, 30, 51, 52, 58, 60].

Successfully identifying a concurrency bug, via a tool or by manually examining the code, does not, however, necessarily mean that a correct fix for it is immediately apparent to the developer. Even worse, by eliminating a data race between memory accesses to the same location, it is not uncommon to introduce violations of other crucial properties, such as introducing a potential deadlock between threads.

Automated program repair (APR) [25] is an emerging suite of technologies for automatically fixing bugs via search, semantic reasoning and learning. In the area of concurrent programs, APR has been employed to address the problem of maintaining multi-threaded software and reducing the cost of migrating sequential code to a concurrent execution model [29, 32, 33, 36–38, 42, 43]. Some tools eschew the issue of detecting bugs, focusing exclusively on repair techniques and assuming that the bug descriptions are already available, e.g., from a bug tracker or in a form of a dynamic execution trace [32, 33, 41–43]. Other tools rely on their own approaches for bug detection, based on run-time analysis or bounded model checking, before continuing with the generation of patches in their specific way [29, 35–38]. However, these tools' reliance on dynamic analyses or bounded model checking for bug detection poses significant challenges to their adoption at large. First, it makes it problematic to integrate APR into everyday development process with low friction, as the developers are required to provide inputs for dynamic executions or structure their tests accordingly, to enable bug detection in the first place. Second, the lack of modularity prevents them from providing incremental feedback to the programmers in the style of continuous integration (CI).

In this work, we address these challenges by describing the first approach to perform APR for concurrency using *static program analysis*. Studies on static analysis usage at Meta [17] and Google [57] show that a developer is roughly 70% likely to fix a bug if presented with the issue at compile time, as compared to 0% to 21% fix attempts when bug reports are provided for checked-in code. The use of static analysis for concurrency bug detection in industry is motivated by its success in catching bugs at scale while minimising *friction* (i.e., adoption effort) and providing *high signal* (i.e., useful bug reports) [7, 9, 14]. The low false positives rate of such analyses is thanks to the design choice to focus on the most common, *coarse-grained*, concurrency (scoped locks and Java's `synchronized` blocks). Restricting the class of analysed concurrent programs this way makes accurate static detection of data races tractable, leading to accurate bug reports on industrial code. Scalability is achieved by making the analysis *compositional* [10]: individual program components (e.g., classes and methods) are analysed in a bottom-up, divide-and-conquer fashion and abstracted as *summaries*. Summaries contain relevant information about the underlying code, which does not have to be re-analysed again. Furthermore, this design favours modularity: the analysis can be executed incrementally, providing nearly instant feedback on recent code changes to the developers.

Our novel approach to APR for concurrent programs builds on a compositional static analysis for data race detection. As the result, our repair tool captures and effectively navigates the fix space for data-race repair. The two main technical challenges we overcome in this work are (a) enhancing a state-of-the-art static analysis for concurrency to collect sufficient information to produce correct concurrent patches efficiently and at scale, and (b) devising a family of algorithms that construct concurrency fixes from the code summaries produced by the augmented analysis. Building on RacerD, an industry-grade static concurrency analyser by Meta [10], our tool, called Hippodrome,[1] implements an automatic repair procedure for *data races* in concurrent Java programs.

[1]Historically, Hippodrome was a place where chariot races were decided.

```
1 public void run() {
2   if (getError() == null) {
3     try {
4 //   synchronized (this) { -- Second (correct) fix (1/2)
5       if (read) {
6         nBytes = getSocket().read(buffers, ...);
7         updateLastRead();
8       } else {
9         nBytes = getSocket().write(buffers, ...);
10        updateLastWrite();
11      }
12 //   }                    -- Second (correct) fix (2/2)
13 // More code
14 }
```

```
// First (faulty) fix and its commit message
- public void run() {
+ public synchronized void run() {
```

Add sync when processing asynchronous operation in NIO.

The NIO poller seems to create some unwanted concurrency, causing rare CI test failures [...] It doesn't seem right to me that there is concurrency here, but it's not hard to add a sync.

Fig. 1. A data race in Apache Tomcat and its fixes.

The static approach endows our technique with several advantages. First, it often requires *no additional input* from the users besides the program itself—enabling smooth integration with the CI workflow. Second, our approach enjoys the underlying analysis modularity, producing fixes in a matter of seconds, thus, *scaling to large real-world codebases*, and allowing for incremental code processing. Finally, the soundness guarantees of the analysis (that hold under certain side conditions), extend to the produced patches: by re-running the analysis on the repaired code we ensure the *correctness of the suggested fixes*: the fixed program satisfies both data-race freedom and deadlock freedom.

To summarise, this work makes the following contributions:

- *Concurrency repair from analysis specifications.* We present a series of algorithms that take the summaries produced by a static analysis for concurrent code and turns them into suggestions for possible fixes, thus delivering the first modular program repair procedure for data races based on a static analysis for concurrency. The design of our patch-generating algorithms addresses a number of pragmatic concerns, minimising the amount of synchronisation to be added for eliminating races, while avoiding deadlocks (Sec. 4).
- *Data race repair tool.* We make the implementation of our approach in a tool called HIPPODROME publicly available [4] for experiments and extensions.
- *Extensive evaluation.* We evaluate HIPPODROME on a number of micro-benchmarks used by related tools for concurrent program repair [38], as well as on two popular large open-source projects (Sec. 5.2).

2 MOTIVATION AND OVERVIEW

In this section, we motivate and outline our approach for concurrency repair based on a compositional static analysis.

2.1 Concurrency Bugs in the Wild

To set the stage for our motivational case study of using a static analysis to detect real concurrency issues, consider the example in Fig. 1. The figure tells a curious story of a data race in the codebase of the Apache Tomcat project. The bug in this code snippet originates from the corruption of buffers; two threads may simultaneously read and write into buffers from the socket. The left-hand side of the figure shows the faulty commit with a non-fix, which has been first attempted in order to remedy the issue. As the enclosed commit message makes evident, this bug was rarely observed, and the developer assigned to fix it was unable to understand the exact root cause of the problem. To make the situation worse, the developer simply made the whole method `run` synchronised. This "fix" removed all concurrency whatsoever

as `run` is the entry method for threads, which all would be forced to run sequentially now. A closer look at the commit message reveals the lack of awareness regarding the `synchronized` primitives. A correct fix by the same developer is shown in the comments.

While this is just one example of a concurrency issue with a wrong human-proposed solution and a simple correct one, there have been multiple instances in the past where an incorrect fix caused other severe problems, e.g., by introducing deadlocks. Thus, there is a need for reliable automated fixes that can guide the developers towards the correct fix. For example, in this bug, an automated fix suggestion (similar to the correct commit) could have helped to avoid the wrong fix.

What gives us hope that there exists a way to engineer an APR procedure for a large class of real-world concurrency bugs is the following observation. The valid fix in Fig. 1 is pleasantly *simple*: it just wraps the subject of the data race into Java's `synchronized` block, thus introducing a necessary mutual exclusion region in the otherwise parallelisable implementation. In fact, it is so simple that we might hope to discover it automatically.

2.2 Dynamic Analysis-based Bug Detection

To demonstrate why automatically characterising concurrency bugs in a repair-friendly way is technically non-trivial, consider the example in Fig. 2, taken from a suite of buggy programs [1] used to showcase tools for concurrent bug detection and APR [27, 38]. Only relevant code parts involved in a data race are shown. In this example, threads may withdraw or deposit in multiple accounts. However, there is a data race in both methods (between line 22 and 24, line 28 and 30). While one thread may read the balance (line 28), another may modify it (line 30). The data race is present in both methods, but the testing thread only checks for the *deposit* method (line 34). Thus, tools relying on test-based bug detection would miss the race in `withdraw`. While test harness could be improved, concurrency bugs in large-scale projects often go undiscovered due to the scheduling problem's intractability. For instance, PFix [38], a recent tool for concurrent APR, uses Java PathFinder (JPF) [65] for bug localisation, but unfortunately, JPF fails to detect the bug in Tomcat (Fig. 1) as it does not scale well to large programs.

2.3 Overview of our Static Approach

We next list the principles that make static analysis suitable for fault detection in large code. First, the class of bugs should be well-defined on the premise that it is natural to give up on detecting *all kinds* of concurrency bugs and focus solely on one, e.g., *data races* as per the current work. Second, reducing the number of false alarms is crucial even when this entails giving up on the soundness (or framing soundness *wrt.* a set of assumptions), a compromise developers accept. Third, to make an interprocedural analysis scale up, it is important for it to be compositional, where the analysed units of code (i.e., methods) are ascribed *summaries*, allowing to re-analyse modified code parts incrementally. For data race detection, it suffices to summarise the *memory accesses* and the *corresponding locks* held at the access sites—thus, achieving compositionality and, hence, scalability.

Unfortunately, the textual bug reports provided by static analysis are not immediately amenable for program repair as they contain too little contextual information. To remedy this, we process the internal summaries of the analysis, extracting the necessary information from them, while solving two key challenges. The first challenge is the *selection of a suitable lock*. A data race is avoided by protecting the affected memory access via the same lock object. However, choosing this lock statically is not trivial: there might be different locks to choose from, or none at all, as is the case in Fig. 2. Besides, new locks should not introduce deadlocks. Second, handling the *scope of the synchronisation* is equally

```
1  public class CustomerInfo {
2    private Account[] accounts;

     // More fields and methods

21   public void withdraw(int accountNumber, int amount) {
22     int temp = accounts[accountNumber].getBalance();
23     temp = temp - amount;
24     accounts[accountNumber].setBalance(temp);
25   }
26
27   public void deposit(int accountNumber, int amount) {
28     int temp = accounts[accountNumber].getBalance();
29     temp = temp + amount;
30     accounts[accountNumber].setBalance(temp);
31   }
32 }
33
34 public void run() { ci.deposit(1, 50); } //Testing Thread
```

```
1  public class Account {
2
3    private int balance;
4
5   public int getBalance() {
6    return balance;
7   }
8
9   public void setBalance(int balance) {
10   this.balance = balance;
11  }
12 }
```

Fig. 2. A data race example from a standard suite [38].

important. Multiple locks may be combined into a single one, producing a concise patch. At the same time, excessive locking inevitably hurts parallelism.

Fig. 3 offers a bird's-eye view of HIPPODROME, our approach to data race patch generation for Java programs. An input program is first statically analysed for data races (Fig. 3, top). If the program contains data races, HIPPODROME progressively translates the output of the analysis into a patch by passing it through sequence of Intermediate Representations (IRs). First, the access summaries and bugs collected from the analysis are merged into a set of bugs (IR1), which are then clustered to support fixes across multiple related bugs (IR2). We design a DSL to encode the patches (IR3) before actually generating and applying them to the input files (IR4). The patches are repeatedly validated by the analysis (the back-link in Fig. 3) for the absence of deadlocks until no more alarms are raised, in which case the bug is considered fixed and the patch is applied, if an automated mode is chosen (bottom left). In an interactive mode (bottom right), the fully validated patches are suggested to the user via an IDE.

We detail our choice of static analysis for fault detection and the algorithmic approach to patch synthesis in the following sections, highlighting what are the abstractions required to connect these two components.

3 STATIC ANALYSIS PRELIMINARIES

The analysis for fault detection lies on top of RACERD [10], an open-source data race detector developed at Meta. RACERD abides to the principles described earlier: it is compositional and it has been empirically demonstrated to provide high signal. The synchronisation primitives are limited to scoped locks and Java's `synchronized` blocks (i.e., so-called *coarse-grained* concurrency) ignoring, e.g., atomic Compare-And-Set and any *fine-grained* synchronisation. This limitation turned out to be an advantage, since a specialised analysis reduces the number of false positives while performing well for industrial code where coarse-grained concurrency is the norm [10].

According to textbook definitions, a race is caused by concurrent operations on a shared memory location, of which at least one is a write [26]. Data races in object-based languages, such as Java, can be described conveniently in a static sense (i.e., without referring to runtime program state), in terms of the program's syntax, rather than memory. This can be achieved via *syntactic access paths* [34], which serve as program-level "representatives" of dynamic operations with

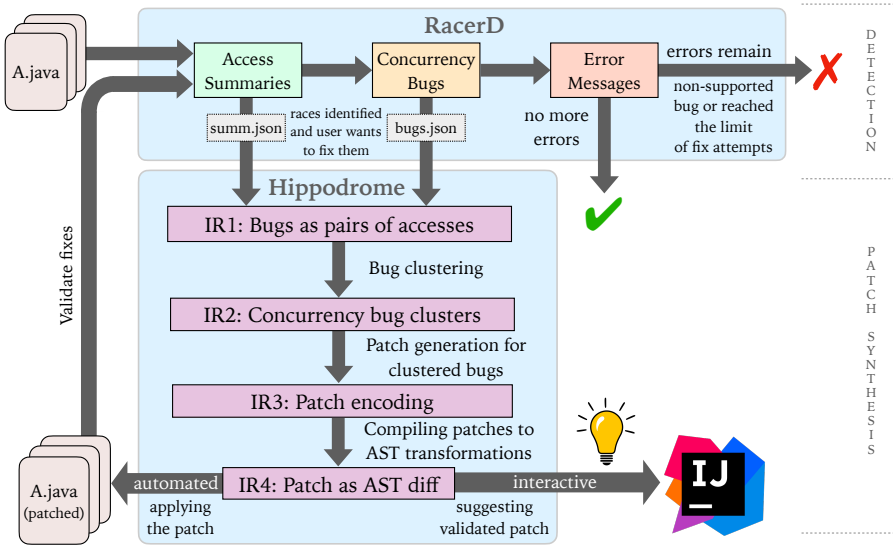


Fig. 3. A summary of HIPPODROME workflow.

memory. An access path is a base variable followed by a sequence of fields, i.e. field-dereferencing chains $x.f_1.\ldots.f_n$, where x is either a variable or `this`. For the example in Fig. 2, `Account.this.balance` is the syntactic access path to the common resource that both `Account.getBalance` and `Account.setBalance` are accessing. For brevity, henceforth, we omit the class-name prefix on variables. We can now define a syntactic data race as follows:

DEFINITION 1 (SYNTACTIC DATA RACE). *Program statements s_1 and s_2 form a* data race *if two* access paths, *π_1 and π_2, pointing to the same memory location, are identified in s_1 and s_2, respectively, such that: (a) executing s_1 and s_2 concurrently, they perform two concurrent operations on π_1 and π_2, at least one of which is a write, and (b) the sets of* locks *held by s_1 and s_2 at that execution point are disjoint.*

The rest of this section briefly describes the abstract domain on which vanilla RACERD operates (without our enhancements which will be detailed in the next section), and how data races are detected. The analysis first conservatively detects Java classes that can be used in a concurrent context, by checking for `@ThreadSafe` annotations, locks, the `synchronized` keyword, or if the instances of those classes appear in the scope of a thread's `run()` method.[2] Subsequent stages work on the premise that *any method* of such a class instance may have data races with *any other method*, including itself. The main routine operates on individual methods, deriving their *summaries*—exemplified below. The analysis is compositional, as those summaries are later used for analysing the code of the caller methods. Finally, the analysis examines method summaries pair-wise in an attempt to identify data races according to Def. 1.

To identify potential races on access paths within concurrently invoked methods of (the same instance of) a class C, the analysis infers those methods' summaries comprised of *access snapshots*. An access snapshot contains mutual exclusion information used for identifying possible data races. Fig. 4 shows the analysis's abstract domain. An access snapshot a is a tuple $\langle \pi, k, L, t, o, \tau \rangle$, where π is the access path, k is a read/write indicator, L abstracts the locks protecting

[2]If a class intended to be used concurrently does not feature any of those "indicators", a developer might have to annotate it explicitly as `@ThreadSafe` to help the analysis.

access paths	$\pi \in \text{Path}$	=	$(\text{Var} \cup \{\texttt{this}\}) \times \text{Field}^*$
trace	$\tau \in T$	=	$(\text{Class} \times \text{Meth})^*$
locks	$L \in \mathcal{L}$	=	$\mathbb{N}$
concurrent thds	$t \in \mathcal{T}$	=	NoThread $\sqsubset$ AnyThreadButMain $\sqsubset$ AnyThread
ownership value	$o \in O$	=	OwnedIf$(\mathbb{N}) \sqsubset$ Unowned
access type	$k \in \{\mathbf{rd}, \mathbf{wr}\}$		
snapshot	$a \in \mathcal{A}$	=	$\langle \pi, k, L, t, o, \tau \rangle$
summary	$A \subseteq \mathcal{A}$	=	$\{a \mid a \in \mathcal{A}\}$

Fig. 4. Abstract domain for race detection.

this access, t and o denote the thread kind and ownership, respectively, and τ is the trace to π and is non-empty for indirect access (i.e., via method calls). For brevity, we refer to elements of this tuple as, e.g., $a.\pi$ for the access path in snapshot a. The thread kind t may take any of the following three abstract values which form a partial order: NoThread — to denote that the current access snapshot belongs to a procedure which cannot run concurrently with any other threads, AnyThreadButMain — when the access snapshot (belonging to the main thread) may be executed in parallel with background threads, or AnyThread — when the current access snapshot belongs to a procedure running on a thread that can interleave with any other thread.

A set A_m of all snapshots collected for a method m is, therefore, an *over-approximation* of all runtime heap accesses reachable from m via the corresponding syntactic access paths, as well as of the corresponding locking patterns.

For the code in Fig. 2, the snapshots collected by the analysis for the two `CustomerInfo` methods are as follows (only showing the problematic access snapshots):

$$A_{\texttt{withdraw}} = \left\{ \begin{array}{l} a_{22} : \langle \texttt{this.accounts[].balance}, \mathbf{rd}, 0, \text{AnyThread}, \text{Unowned}, \{\texttt{Account.setBalance()}\}\rangle, \\ a_{24} : \langle \texttt{this.accounts[].balance}, \mathbf{wr}, 0, \text{AnyThread}, \text{Unowned}, \{\texttt{Account.getBalance()}\}\rangle, \ldots \end{array} \right\}$$

$$A_{\texttt{deposit}} = \left\{ \begin{array}{l} a_{28} : \langle \texttt{this.accounts[].balance}, \mathbf{rd}, 0, \text{AnyThread}, \text{Unowned}, \{\texttt{Account.getBalance()}\}\rangle, \\ a_{30} : \langle \texttt{this.accounts[].balance}, \mathbf{wr}, 0, \text{AnyThread}, \text{Unowned}, \{\texttt{Account.setBalance()}\}\rangle, \ldots \end{array} \right\}$$

$$A_{\texttt{getBalance}} = \left\{ \; a_6 : \langle \texttt{this.balance}, \mathbf{rd}, 0, \text{AnyThread}, \text{Unowned}, \{\}\rangle \; \right\}$$

$$A_{\texttt{setBalance}} = \left\{ \; a_{10} : \langle \texttt{this.balance}, \mathbf{rd}, 0, \text{AnyThread}, \text{Unowned}, \{\}\rangle \; \right\}$$

The snapshot for line 22 states that there is a *read* access to a heap location pointed to by `this.accounts[].balance`. Moreover this memory location may be accessed by any other thread (AnyThread) unrestrictedly since there is no lock protecting it (denoted via 0), and it is not owned by, i.e. not local to, any thread (Unowned). The final element in the tuple represents the access's *trace* call, meaning that it is an indirect access done by calling the method `setBalance()` of the class `Account`.

Conflicts are detected in our running examples by pairwise checking the snapshots in $A_{\texttt{withdraw}}$ and $A_{\texttt{deposit}}$ against each other. This check concludes that every pair of paths refers to the same heap address and they may be accessed concurrently. Moreover the snapshot pairs (a_{22}, a_{24}), (a_{28}, a_{30}), (a_{22}, a_{30}) and (a_{24}, a_{28}) involve at least one write.

The next section shows how to use such summaries for data race detection and, subsequently, for patch inference.

4 INFERRING AND APPLYING PATCHES

This section describes the enhancements we brought to the static analysis, what is the space of solution, and how we finally infer and apply the patches.

4.1 Fault Detection

We first motivate and describe the enhancements of the static analysis for fault detection.

4.1.1 Bug reporting strategy. To reduce the number of warnings which may be overwhelming for the developer, RacerD only reports at most one pair of read-write conflicting snapshots per access location. For instance, it will report (a_{22}, a_{24}), (a_{28}, a_{30}) as potential bugs for the example in Fig. 2, although as mentioned in the previous section there are two more problematic pairs, namely (a_{22}, a_{30}) and (a_{24}, a_{28}). This is a sensible design choice for manual repair where the developer is directed progressively to new conflicts as she repairs old ones and re-runs the analysis. Automatic repair on the other hand can be instrumented to process multiple conflicts at once. A complete view of the bugs allows for a better patch generation strategy, hence we modify the analysis to log all possible bugs.

4.1.2 Enhanced lock domain. A naïve (and wrong) fix for the race in Fig. 2 is to wrap lines 22-24 in `synchronized(m1){...}`, and lines 28-30 in `synchronized(m2){...}`, where `m1` and `m2` are freshly created mutexes such that $\texttt{m1} \neq \texttt{m2}$. The access snapshots would be the same except for the locks component, which would change from 0 to 1 in each of the four tuples, making RacerD erroneously infer the *absence* of a race, due to the unsoundness caused by non-conservative interpretation of its abstract domain $\mathcal{L}$ for locking (*cf.* Fig. 4). This is because its lock domain is defined to only track the *number* of locks used to protect an access in the actual code, but not their identity. As a remedy to this unsoundness, we change the abstract domain of locks $\mathcal{L}$ from natural numbers to the *powerset of paths* (i.e., the new $\mathcal{L} \triangleq \wp(\text{Path})$), where the identity of each involved lock is abstracted by its syntactic access path. The new domain affords better race detection, while offering crucial information for the repair purposes: *which* exact locks are taken at the location of a race. In the remainder of the paper, we will keep referring to the enhanced analysis implementation as RacerD.

4.1.3 Static Race Detection from Summaries. Following the definition of a syntactic data race between two program statements in Sec. 3, we define a data race between two access snapshots as follows:

$$
\begin{array}{lll}
\textbf{let } \mathit{race}(a_1,\ a_2)\ = & & \\
\quad a_1.\pi = a_2.\pi & & \textit{(same path)} \\
\wedge\ (a_1.k = \textbf{wr} \vee a_2.k = \textbf{wr}) & & \textit{(at least one write)} \\
\wedge\ a_1.L \cap a_2.L = \emptyset & & \textit{(disjoint locks)} \\
\wedge\ a_1.t \sqcup a_2.t = \text{AnyThread} & & \textit{(possibly concurrent)} \\
\wedge\ (a_1.o\ =\ \text{Unowned} \wedge a_2.o\ =\ \text{Unowned}) & & \textit{(shared memory)}
\end{array}
$$

Since a program statement may contain zero, one or more snapshots, it results that if there is a race relation between two different access snapshots, say a_1 and a_2, each belonging to two different statements, $a_1\ \in s_1$ and $a_2 \in s_2$, then there is a data race between s_1 and s_2 as well. Moreover, two methods, mth_1 and mth_2 manifest a race if there exist snapshots $a_1 \in A_{mth_1}$ and $a_2 \in A_{mth_2}$ such that $\mathit{race}(a_1, a_2) = \text{True}$. Two accesses are thus guaranteed to not form a data race if they are both protected by the same lock or performed by the same thread.

In addition to the binary data race formulation, we define a unary relation over access snapshots as follows:

$$
\begin{array}{ll}
\textbf{let } \mathit{unprotected\_write}\ a\ = & \\
\quad a.k = \mathbf{wr} & \textit{(write operation)} \\
\quad \wedge\ a.L = \emptyset & \textit{(unlocked)} \\
\quad \wedge\ a.t = \text{AnyThread} & \textit{(possibly concurrent)} \\
\quad \wedge\ a.o = \text{Unowned} & \textit{(unowned resource)}
\end{array}
$$

A method *mth* is said to be *unsafe* when there exists $a \in A_{mth}$ such that $\mathit{unprotected\_write}(a) = \text{True}$. In the next section, we show how the enhanced domain along with the definition of a data race and unsafe method suffice to derive a patch, and subsequently a fix for the example from Fig. 2.

4.2 Searching for Repairs

Let us revisit the running example in Fig. 2 with synchronisation missing. In particular, let us focus on the bug described by the pair (a_{22}, a_{24}). With no lock whatsoever to protect these accesses we are in the situation where a number of fixes can be applied in a possibly automated way:

- We can protect both accesses by individually wrapping the affected lines into `synchronized(this)` statements. While straightforward, relying solely on the built-in lock, i.e., on `this`, may introduce unnecessary blocking and synchronisation overheads, particularly when there is no reason to prevent the interleaving of threads which share different resources. Although commonly used, best practices in concurrent programming recommend using the built-in lock with care since it "forces JVM implementers to make tradeoffs between object size and locking performance" [23].
- Alternatively, we can create a fresh object to add the two `synchronized` statements on it, thus avoiding contention with other `synchronized` statements in this class. While valid, this approach may generate too many locks, especially for accesses involved in multiple data races.
- Finally, we can annotate the shared variable as `volatile`. By declaring a variable as volatile all writes to it will be written directly into the main memory. Similarly, all reads of the variable will be read directly from main memory. In other words, all threads can access the shared variable with the newest, up-to-date value. While many developers' favourite solution, best practices only recommend it when the resource is shared for reading by multiple threads, but written to by only one [23]—a piece of knowledge beyond the reach of static analysis which detects pairs of program components that may be executed concurrently but not the number of threads to be spawned for a particular code. Simply put, `volatile` is used to ensure the visibility of shared variables in multithreaded environments, and not their mutual exclusion. Furthermore, this approach is not straightforward to apply for array entries.

This short, informal and non-exhaustive analysis of synchronisation possibilities highlights the challenges in choosing a strategy for automatic patch generation, with the observation that it is often a compromise between simplicity of a fix and avoiding over-synchronisation. A resource-aware lock choice heuristic would make minimal system calls, would avoid frequent context switches, and would aim for reduced memory-synchronization traffic on the shared memory bus. We believe that designing such heuristics is a challenging problem, worth pursuing as a stand-alone research topic in future. In this work, whose focus is on the synergy between static analysis and data races repair, we settle for a perhaps non-ideal, but carefully crafted solution in choosing the lock objects: since over-synchronisation is probable when using intrinsic locks on programs which manipulate multiple resources, and the deadlocks with such a strategy would be fatal [23], we instead focus - whenever feasible - on a more *prudent choice of external mutexes for*

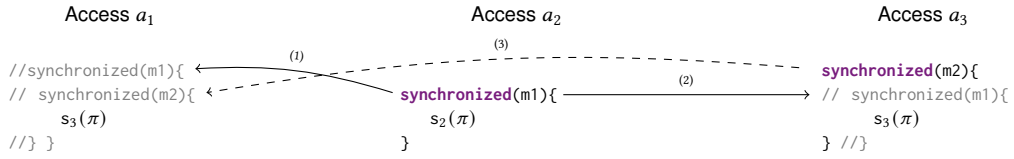


Fig. 5. Three data races and their naïve, order-sensitive fixes.

manipulating `synchronized` *blocks*. We chose to synchronise on the built-in object only when `this` has already been used by the developer to partially protect the access to a resource which may be non-exclusively updated.

Moving further, although the above-mentioned solutions are common in APR or manual repair, none tackles the root cause of the race, namely that the accesses to `this.accounts[].balance` are unprotected in the `Account` class (which is possibly used somewhere else, too). This observation shows that treating a concurrency bug as a standalone entity would deprive one from opportunities to produce high-quality fixes for families of related issues. To regain the holistic view on the origins and the implications of data races, we propose a mechanism to *cluster related bugs* allowing us to find a suitable common patch.

4.3 Patch Inference

The main pipeline of Hippodrome transforms summaries and bug reports obtained from the bug analysis (Fig. 3, top) to a sequence of Intermediate Representations (IRs), eventually obtaining patch candidates for the discovered concurrency bugs. We proceed to describe the involved IRs and the corresponding algorithms for producing them.

IR1. A bug is reported either as a pair of access snapshots (in the case of Read/Write races) or as a single snapshot (in the case of unprotected writes):

$$\text{bug } b \in \mathcal{B} = \{\langle a \rangle \mid \mathit{unprotected\_write}(a) = \text{True}\} \cup \{\langle a_1, a_2 \rangle \mid \mathit{race}(a_1, a_2) = \text{True}\}$$

Given a Read/Write bug b, a straightforward strategy is to pick any one of the locks in $b.a_1.L \cup b.a_2.L$ if any exists, or create a fresh one otherwise. For instance, for the bug (a_{22}, a_{24}) in our example (Fig. 2), since $a_{22}.L \cup a_{24}.L = \emptyset$, we would create a new object, say `m1`, and insert the appropriate `synchronized` blocks. Similarly, we could fix the (a_{28}, a_{30}) via a fresh mutex, say `m2`. For the remaining two bugs which involve inter-method accesses we would create new mutexes, or choose from the freshly created ones. Either way this solution already creates nested synchronisation blocks breaking maintainability and increasing the chance of deadlocks.

Example 1. To better highlight these issues consider in Fig. 5 a simplified, yet more general instance of the above situation depicting three program statements sharing the same resource π, where at least two of the statement involve a write operation on π. Initially, s_2 and s_3 are protected by mutexes `m1` and `m2`, respectively. The analysis would report (a_1, a_2), (a_2, a_3) and (a_1, a_3) as data races. Zooming individually into each bug could lead to fixes where `synchronized` statements are introduced as suggested by the arrows. The arrows are annotated with the order in which the fix is generated. Clearly this pseudo-fix is problematic, not only due to the clutter of synchronisations but also because it naïvely introduced a deadlock (dashed arrow). Can we do better?

IR2. To improve patch quality, instead of deriving patches from individual bugs, we will infer them from bug *clusters*. We will do so by analysing *sets* (instead of *pairs*) of snapshots, grouping bugs according to their shared access paths. For a set of bugs B, a cluster C_B is defined as follows:

$$C_B \triangleq \{\, C_B^{\pi} \mid \pi \in \Pi_B \} \quad \text{where} \quad \Pi_B \triangleq \{\, \pi \mid \pi \in \bigcup_{a \in acc(b)} \{a.\pi\}, b \in B\}$$
$$C_B^{\pi} \triangleq \{\, b \in B \mid \forall a \in acc(b) :\ a.\pi = \pi\}$$

where the function *acc* returns the set of snapshots summarising the bug. For the clarity of presentation, we assume that all the access snapshots in a cluster belong to the same Java class (in practice, we refine the above clustering by taking classes into the account). We define a function $cls(a)$ to return the class of an access snapshot a, and generalise it to indicate the class of a bug cluster (since all its bugs share the same access).

For our running example, a cluster of bugs related by their unprotected access to `this.accounts[].balance` is the set

$$\{(a_{22}, a_{24}), (a_{28}, a_{30}), (a_{22}, a_{30})(a_{24}, a_{28})\}$$

IR3. Algorithm 1 infers patches from bug clusters. Patches are encoded using the following simple domain-specific language which supports insertion of `synchronized` blocks, variable declarations, and `volatile` annotations:

$$\begin{array}{lll} \text{action} & act & ::= \text{SYNC}(A, lock) \mid \text{DECLARE}(class, var) \mid \text{VOLATILE}(class, var) \mid \text{NIL} \\ \text{composition} & p & ::= act \mid \text{AND}(act, act) \mid \text{OR}(act, act) \end{array}$$

SYNC(A, *lock*) indicates that the snapshots in A should each be wrapped in a `synchronized` block on mutex *lock*. DECLARE(*class*, *var*) suggests that a new variable *var* (implicitly of type *Object*) should be declared in *class*, while VOLATILE(*class*, *var*) introduces an obligation to annotate *var* with `volatile`. AND is used to compose patch components corresponding to different snapshot accesses of the same bug cluster. OR denotes different patch options for the same access snapshot. NIL is an intermediary action, solely used to ensure composition's well-formedness.

Even though seemingly simple, Algorithm 1 has a few subtle points. First, for each bug cluster, the algorithm derives a patch to ensure that all of its accesses are protected by a common lock. As far as we know, there is no best strategy for choosing this lock when some access paths are already protected. Of several possible strategies applicable in this case, we have settled on choosing a lock that protects the most accesses, thus avoiding nested synchronisation. However, the lock choice strategy can easily be adjusted by choosing how to manipulate the list of locks at line 13. Since a final patch is a composition of sub-patches (line 23), the algorithm could easily support a more advanced lock choice strategy (if one would be proposed in the future) which could introduce multiple mutexes for a cluster of bugs as opposed to a common mutex for all bugs in a cluster as currently proposed.

Another subtle point of the algorithm is protecting the innermost access in the call chain leading to a race, thus solving the root cause of the bug and reducing the number of locations to be synchronised. An alternative solution is to synchronise at the call site. We support this option too, though not as a default. The benefit of this alternative is that it could be tailored to solve simple atomicity violations [47], but those are not our main focus. We discuss in Sec. 4.5 how to derive patches for simple atomicity violations and what are the challenges for tackling the general case.

The recommended patch for the running example is as follows (written in infix order for readability, and ignoring the volatile solution for brevity):

$$\text{DECLARE}(Account, v) \text{ AND } \text{SYNC}(\{a_{getBalance}\}, v) \text{ AND } \text{SYNC}(\{a_{setBalance}\}, v)$$

where $a_{getBalance}$ and $a_{setBalance}$ are the snapshots corresponding to the accesses to `this.accounts[].balance` from `Account`'s `getBalance` and `setBalance`, respectively. In other words, HIPPODROME recommends patching a bug at its source instead of the location of its manifestation.

Algorithm 1: CreatePatchEncodings

```
Input:  a powerset of bugs C
Output: a patch set P
P_C ← ∅
for B ∈ C do
    A ← union of acc(b) for all b in B // snapshots for a cluster B
    locks ← union of a.L for all a in A
    if locks = ∅ then
        // if no locks available, create a new mutex
        var ← fresh variable name
        act ← DECLARE(cls(B), var)
        locks ← {var}
    else
        locks ← order locks according to their frequency (descending)
        act ← NIL
    fixes ← NIL // collects all possible fixes for cluster B
    for lock ∈ locks do
        P ← ∅ // collects patch components
        for a ∈ A do
            a′ ← find the innermost access snapshot in a.τ common to all accesses in A
            // synchronising the statement containing a′ via lock
            p ← SYNC({a′}, lock)
            P ← {p} ∪ P
        patch ← AND(act, AND(P)) // combine patch components
        fixes ← OR(fixes, patch)
    x ← a field subject of race in all bugs of B
    p_B ← OR(fixes, VOLATILE(cls(B), x))
    P_C ← {p_B} ∪ P_C
return P_C
```

IR4. The final step in the Hippodrome pipeline is to translate the encodings from the preceding stage to actual patches. This process is described by Algorithm 2, which produces patches in the following simple language of AST-manipulating actions taking tree nodes as their arguments:

$$
\begin{array}{lrcl}
\text{action} & \widehat{act} & ::= & \text{REPLACE}(from, to, ast) \\
 & & & \mid \text{INSERT\_AFTER}(stmt, ins, ast) \\
 & & & \mid \text{INSERT\_BEFORE}(stmt, ins, ast) \\
\text{composition} & \widehat{p} & ::= & \widehat{act} \mid \text{AND}(\widehat{act}, \widehat{act}) \mid \text{OR}(\widehat{act}, \widehat{act})
\end{array}
$$

The algorithm relies on several auxiliary procedures for declaring variables, adding volatile modifiers, and inserting locks. We omit the definitions of all but the last one, as the rest are straightforward. We note that the only delicate matter for DeclareVariable is to identify whether any of the SYNC accompanying the peer DECLAREs belongs to a static method. If that is the case the declared variable must be annotated as `static`, too. Algorithm 3 defines the insertions of `synchronized` blocks, which might require splitting variable definitions into declarations and initialisations if there are declarations of variables which outlive the scope of the newly introduced block. Choosing to split the variable definitions allows patches with less invasive synchronisation, i.e., smaller sized synchronised blocks.

Algorithm 2: CREATEPATCH

Input: a patch encoding p_0 in IR3 format
Output: a patch as AST diff $\widehat{p}$
$p \leftarrow$ normalise AND such that any SYNC(A_1, *lock*) and SYNC(A_2, *lock*) are merged into SYNC($A_1 \cup A_2$, *lock*) if all the snapshots in A_1 and A_2 belong to the same method
switch p **do**
  case *AND*(act_1, act_2) **do**
    return AND(CREATEPATCH(act_1), CREATEPATCH(act_2))
  case *OR*(act_1, act_2) **do**
    return OR(CREATEPATCH(act_1), CREATEPATCH(act_2))
  case *SYNC*(*A*, *lock*) **do**
    return INSERTLOCK(p)
  case *DECLARE*(*class*, *x*) **do**
    return DECLAREVARIABLE(*class*, *x*)
  case *VOLATILE*(*class*, *x*) **do**
    return MAKEVOLATILE(*class*, *x*)

Algorithm 3: INSERTLOCK

Input: a SYNC(*A*, *lock*) patch
Output: a patch as AST diff $\widehat{p}$
class $\leftarrow$ retrieve AST of *cls*(*A*)
from $\leftarrow$ the closest parent in *class* of a node containing all *a* in *A*
cond $\leftarrow$ true if *from* has variables that outlive the scope of *from*
if *cond* **then**
  decl, *from*′ $\leftarrow$ move variable declarations out of *from*
  // *from*′ now contains only initialisers of the variables
  $\widehat{act}_d$ $\leftarrow$ INSERT_BEFORE(*from*, *decl*, *class*)
  sync $\leftarrow$ wrap a `synchronized` (*lock*) `{}` around *from*′
  $\widehat{act}_i$ $\leftarrow$ REPLACE(*from*, *sync*, *class*)
  return AND($\widehat{act}_d$, $\widehat{act}_i$)
else
  sync $\leftarrow$ wrap a `synchronized` (*lock*) `{}` around *from*
  $\widehat{act}$ $\leftarrow$ REPLACE(*from*, *sync*, *class*)
  return $\widehat{act}$

In our running example, the algorithm inserts a variable declaration, instantiates an object in the `Account` class, and replaces in `getBalance` and `setBalance` the two unprotected accesses to `this.accounts[].balance` with their corresponding accesses protected by the freshly created object. For *Example 1*, the fix produced by HIPPODROME avoids the deadlock by only inserting synchronisations (1) and (2).

The resulting patches are normalised into a "disjunctive normal form", i.e., a disjunction of conjunctions, where each disjunct represents a possible multiline, multilocation patch. When HIPPODROME is used in an interactive mode, the user receives a list of complete patches to choose from. When used in auto-repair mode, the tool favours the SYNC patches with the least number of insertions.

Patch application and validation. The patch application is straightforward, involving changes in the original file's AST. Given a fix OR($\widehat{act}_1$, $\widehat{act}_2$) for a bug cluster B, the patch application strategy chooses to apply patch $\widehat{act}_1$, and stores $\widehat{act}_2$ for subsequent application should the validation phase fail. Once completed, the files are validated by RacerD (back-link in Fig. 3). The analysis *can detect deadlocks*—a feature we actively use to discard patches that might introduce such issues. Even with no deadlock detected, the program may still contain bugs (e.g., data races on a different access path), hence we keep the fix and reiterate through the algorithm to fix the remaining issues, doing so until there are no more bugs left, or the iteration limit (currently 10) is reached. In our case studies, we have never hit that limit. We guarantee to introduce *no new races* due to the fact that we never update existing locks, but only introduce more synchronisation. In Sec. 4.4, we state our correctness claims, enumerating the assumptions under which they hold.

Patch quality. In designing the strategy for generating and automatically choosing patches, we have considered reducing both (a) context switching as well as (b) over-synchronisation. We optimise for (a) by clustering bugs based on their shared access path and by merging lock statements on the same mutex located in the same method (the latter one is optional). We optimise for (b) by avoiding intrinsic locks (i.e., on `this`), and by optimising each bug cluster to be served by mutexes not serving to synchronise any other clusters.

4.4 Correctness and Limitations

Measures of Correctness. It is important to understand what aspects of our approach offer confidence that Hippodrome only catches actual data races, and that the patches produced to repair these bugs are indeed correct. We split this discussion into three parts, one corresponding to the bug detection phase, one to the patch synthesis phase, and one for patch validation.

The correctness of the bug detection (i.e., whether it only reports actual data races) relies on the underlying analysis. Under certain assumptions, RacerD is proven to report *no false positives*—that is, if it reports a data race, then there exists a witness execution for an access path that triggers the reported concurrency bug [24]. Specifically, RacerD assumes that every execution branch may be taken, i.e., the control for conditional statements and while loops is non-deterministic, and that it operates on the concurrency model restricted to balanced reentrant locks and coarse-grained locking. This model has been shown to provide high signal in production code [10].

The correctness of the patch synthesis (i.e., the produced patch repairs the reported race) relies on the fact that all program locations involved in a cluster of bugs are guaranteed to be protected by the same mutex. In other words, Hippodrome wraps all program locations that may concurrently access the same resource in a `synchronized` block of the same object. While our choice to cluster the bugs was instrumental in offering these guarantees and in reducing the chance of a deadlock, patches that deadlock may still be synthesised. Luckily, the patch validation phase prunes away such pseudo-patches. The question now is whether there is any guarantee of producing at least one valid patch. The answer is yes: in the worst-case scenario, when all patches which involve synchronization of existing objects lead to possible deadlocks, Hippodrome synthesises a patch where it creates a fresh object on which to synchronise all locations of interest. Furthermore, choosing to insert the `synchronized` block of the new object as the innermost wrapper in a structure of nested synchronization blocks ensures that the interference with existing mutexes cannot produce a deadlock.

The patch validation phase is guaranteed to not miss any deadlock [3] under two crucial, yet reasonable, assumptions:

[3]The deadlock analysis is proven sound for a language with scoped re-entrant locks, nondeterministic branching, and non-recursive procedure calls [11].

(a) Locking is *balanced*, i.e., lock releases follow a LIFO order. This effectively corresponds to the case of Java's `synchronized`, which only allows lexically-scoped locks.
(b) Lock-acquiring code is located within *non-anonymous* classes since summaries have to be ascribed to methods of named classes. Therefore, deadlocks may be missed when the locking code is inlined in the spawned threads.

Assumption (b) is satisfied by industrial software, which always implements synchronisation *in libraries*, while spawning threads in thread pools. However, micro-benchmarks used to evaluate dynamic analysis-based tools might violate it as they favour concise and self-contained code for tractable dynamic race detection.

To sum up, if HIPPODROME catches a data race, it is guaranteed to synthesise at least one valid patch—one which fixes the race and introduces no deadlocks.

High Recall. Our approach relies on the guarantees of RACERD to achieve a high recall (i.e., to catch *all* data races). The assumptions under which the analysis does not miss races are listed in [10, §8]. Specifically, it assumes that an owned access and its suffixes remain owned throughout the current procedure. Violating this assumption can lead to false negatives if a local object is leaked to another thread *and* if the other thread accesses one of the object's fields. RACERD also missed bugs that violate assumption (b) above or races which involve statements in the body of an exception handler, and that is because the IR on which it operates does not have support for exception handling.

At the time of this submission, the static analysis team at Meta confirmed that both race and deadlock detection were running in production for more than two years *with no false negatives identified* [11].

4.5 A Heuristic for Fixing Simple Atomicity Violations

Both data races and atomicity violations are a consequence of violations in the synchronisation. The former refers to two concurrent accesses on the same memory location causing an inconsistent memory read or update. The latter refers to a thread's two consecutive accesses to a shared memory location interleaving with an unserializable access from another thread. It is important to note that not all program races lead to atomicity violations, and vice-versa, not all atomicity violations contain data races. RACERD is a data race detector, so it comes natural to conclude that HIPPODROME would only be able to fix data races.

Contrary to the fact that HIPPODROME only works over bug reports for data races, it can actually be instructed to fix a restricted category of atomicity violations too. We identified that atomicity violations which also contain data races, may be tackled too as long as the problematic accesses are subsumed by the reported data races and if the consecutive access from a thread to a shared location are performed by the same method. For example, a visual examination of the example in Fig. 2 reveals both the existence of data races (between the update on the `balance` field of class `Account` and any other direct or indirect access to this field - as detailed in Sec. 4.3), as well as atomicity violations (for the following interleavings of statement executions: $22 - 30 - 24$ or $28 - 24 - 30$).

Fixing the data race as HIPPODROME does, by wrapping the accesses to the `balance` field of `Account` class with synchronized blocks on the same object, although correct for the domain it serves, it does not fix the atomicity violations in the `CustomerInfo` class. To detect and fix the atomicity violations, we could inspect again the access summaries of methods `withdraw` and `deposit` of class `CustomerInfo`, notice that both methods have unprotected accesses to the same resource, and protect all four statements with the same mutex lock. Wrapping each statement in a synchronized block would still only protect from data races, but not atomicity violations. To further fix this issue, the synchronized block protecting the first access in each method should fold over the entire code between the first and the last access to the

shared resource in the same method. The new patch fixes both the data race (since all the invocations of the methods in `Account` are protected in `CustomerInfo`) as well as the atomicity violations:

```
1   public class CustomerInfo {
2     private Account[] accounts;
3     Object obj1 = new Object();

      // More fields and methods

21    public void withdraw(int accountNumber, int amount) {
22      synchronized{obj1} {
23        int temp = accounts[accountNumber].getBalance();
24        temp = temp - amount;
25        accounts[accountNumber].setBalance(temp);
26    } }
27
28    public void deposit(int accountNumber, int amount) {
29      synchronized{obj1} {
30        int temp = accounts[accountNumber].getBalance();
31        temp = temp + amount;
32        accounts[accountNumber].setBalance(temp);
33    } }
34  }
```

To support this feature, line 19 in Alg. 1 should be modified to search for the outermost access snapshot in $a.\tau$ common to all accesses in the cluster of bugs (bug manifestation), instead of the default innermost which aims to fix the data race at its source. Creating the patch as per Alg. 2 is already designed to fold over all the statements in a method, hence no other modifications need to be made to the existing algorithms.

Since this solution for fixing atomicity violations is formulated as a heuristic, we are not offering any guarantees, hence we do not claim it as a contribution, and instead leave it as a formal investigation for future work. It is meant to highlight the importance of having a modular design which allows us to easily replace existing functionality or add new features in order to improve the quality of the final patches. A general atomicity violation solution would have to take into account a thread's consecutive cross-method accesses to shared resources. This entails further knowledge about the program's logic and support for more fine-grained synchronization primitives. The latter simply involves the extension of current DSLs; however, the former is more difficult to tackle and it requires a different kind of static analysis for atomicity violations—one which can infer inter-procedural atomicity requirements.

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation

As per Fig. 3, our approach is implemented in a framework comprising the following components: *bug detector*, *patch synthesiser* and *application*, and *validator*. For *bug detection* and *validation*[4], we build on RacerD (implemented in OCaml) extending its lock domain (Sec. 4.1.2) and the bug reporting strategy (Sec. 4.1.1). RacerD already stores concurrency bug reports in log files, but not the method summaries, for which we had to add support. The interaction with Hippodrome's core components is achieved via JSON files. Hippodrome's *patch synthesiser and application* are implemented in Scala, comprising about 2.5k lines of code. The synthesiser expects as input a set of bugs and a set of method summaries, and

[4]While working on the validation phase, we discovered a bug in how the locks were treated in the deadlock detector which we have fixed in our analysis and responsibly reported it to the RacerD team.

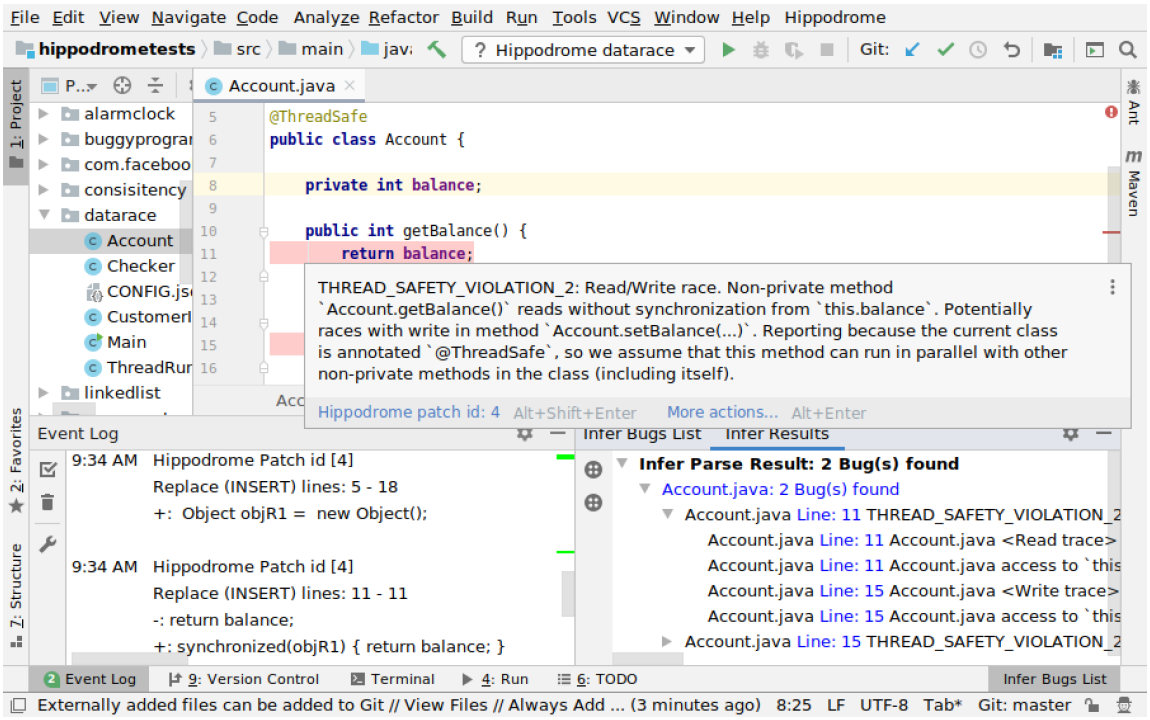


Fig. 6. Using HIPPODROME in an interactive mode.

then proceeds to construct all the intermediate representations (Sec. 4.3). Although we have only considered patches which involve inserting `synchronized` statements or `volatile` annotations, staging the patch synthesis process into IRs allows future works to add support for other synchronization primitives by simply extending these IRs correspondingly. Manipulating the Java files when constructing patches (Sec. 4) is done via ANTLR v4.

When HIPPODROME runs in automatic mode, multiple patches may be generated for a single cluster of bugs. To decide which patch is the best fit, we have added parametrised support for a cost function to choose the fix candidate. We have implemented a simple cost function which measures the number of statements which need to be updated for a fix, and choose the least intrusive fix (smallest number of updates). Users of our framework could explore other application specific cost functions if the default cost function is not satisfactory. Only the best candidate fix is validated in this mode.

When HIPPODROME runs in interactive mode, all possible fixes are validated before being presented to the user which then chooses the fix it considers as best fit. For a better UX experience, we have integrated HIPPODROME into IntelliJ IDEA as a plug-in. The screenshot in Fig. 6 demonstrates a report produced for a data race in the `Account` class, as well as a patch suggested by HIPPODROME (bottom left window), which will create a new mutex object `objR1` and use it for synchronising accesses to `Account`'s field `balance`.

Since we have kept a clean separation between the bug analysis and the implementation of the patch generation mechanism, the framework could easily benefit from the advancements in data race detection, e.g. [40], as long as the artefacts of these new technologies can discharge to JSON files the analysis results in the format described in Fig. 4.

Our tool HIPPODROME is publicly available as a Docker image [4], or as an open-source project [5].

Table 1. Results on subject apps by PFix and Hippodrome. All times are given in seconds. Det. means "detection". # Iter refers to the number of times RacerD was invoked within a single run of Hippodrome

Subject Apps			PFix			RacerD Det.	Hippodrome			
Program Name	**LOC**	**From Benchmark Suites**	**Fix Status**	**Det. Time**	**Fix Time**		**Fix Status**	**Det. Time**	**Fix Time**	**# Iter**
account	102	PECAN, JCTB-Toy, PFix	Success	22.7	238.75	Fail	N/A	N/A	N/A	N/A
accountsubtype	138	PFix	Success	29.4	21.4	Fail	N/A	N/A	N/A	N/A
airline	51	JCTB-Toy, PFix	Success	8.35	16.2	Success	Success	1.34	0.71	2
alarmclock	206	JCTB-Toy, PFix	Fail	10.75	N/A	Success	Success	4.51	2.07	4
allocation vector	114	JCTB-Toy	Fail	N/A	N/A	Success	Success	2.42	0.73	2
atmoerror	48	PFix	Success	7.3	5.95	Success	Success	1.45	0.43	2
buggyprogram	258	PECAN, PFix	Success	9.45	33.55	Partial	Partial	1.61	2.87	2
checkfield	41	PFix	Success	7.15	9.8	Success	Success	1.24	0.8	2
consisitency	28	PFix	Success	6.75	9.95	Success	Success	2.3	0.87	3
critical	56	PECAN, JCTB-Toy, PFix	Success	15.4	14.1	Success	Success	5.8	1.15	6
datarace	90	PFix	Success	8.1	51.15	Success	Success	1.28	0.28	2
even	49	PFix	Success	7.25	91.15	Success	Success	2.14	0.36	2
hashcodetest	1,258	PFix	Success	8.45	7.45	Success	Success	4.62	1.65	4
linkedlist	204	PECAN, JCTB-Toy, PFix	Success	7.95	35.25	Success	Success	0.97	1.11	2
log4j	18,799	JCTB-Toy, PFix	Success	22.9	20.35	Success	Success	1.49	1.68	2
Manager	130	PECAN	Fail	N/A	N/A	Success	Success	2.54	1.43	3
mergesort	270	PECAN, PFix	Fail	17.95	N/A	Success	Success	1.07	2.87	3
pingpong	130	PFix	Success	25.2	23.05	Success	Success	3.67	1.48	4
ProducerConsumer	144	PFix	Fail	16.0	N/A	Success	Success	4.61	1.61	6
reorder2	135	JCTB-Toy, PFix	Success	7.7	11.9	Success	Success	1.32	0.58	2
store	44	PFix	Success	7.2	5.85	Success	Success	1.22	0.29	2
stringbuffer	416	PECAN, PFix	Success	7.0	22.2	Fail	N/A	N/A	N/A	N/A
wrongLock	73	JCTB-Toy, PFix	Success	7.15	5.9	Success	Success	1.24	0.36	2
wrongLock2	36	PFix	Success	7.3	16.4	Success	Success	1.39	0.94	2

5.2 Evaluation

We empirically evaluated Hippodrome's effectiveness in producing high-quality fixes for Java data races. Experiments were designed to answer the following Research Questions:

RQ1: How does Hippodrome compare to the state-of-the-art repair tools in terms of performance and efficacy?

RQ2: What is Hippodrome's performance on large projects and how do the patches it produces compare to developers' manual fixes in those projects in terms of quality?

All our experiments were done on a commodity laptop with 16 GB RAM and an 8-Core Intel 2.3GHz CPU running macOS.

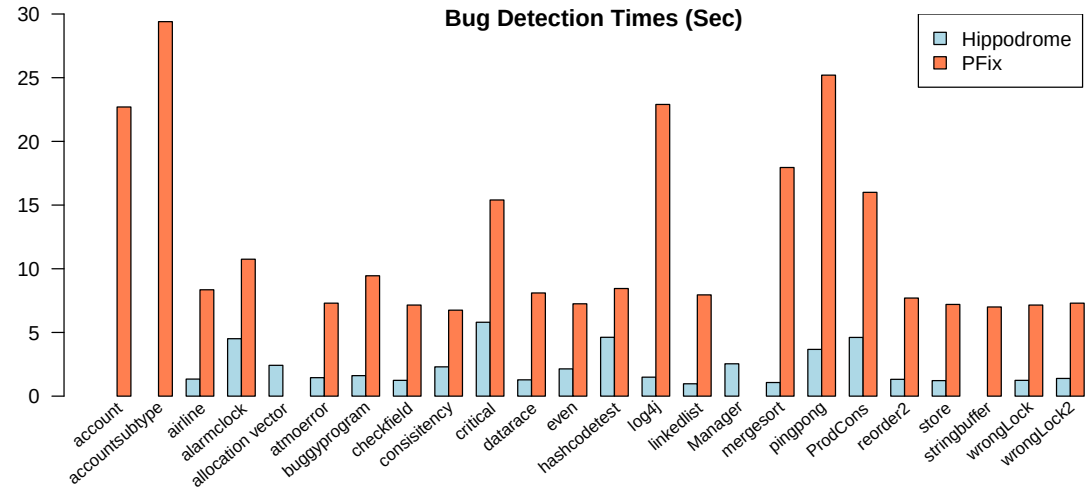
5.3 RQ1: Comparison to the State of the Art

Multiple tools [12, 32, 33, 38, 42] have been proposed to repair concurrency bugs. Of those, PFix [38] is the most recent concurrency bug-fixing tool for Java programs, shown to significantly outperform the previous state-of-the-art, namely Grail [42], in terms of efficiency, correctness and patch quality. Moreover, similar to our approach, PFix also targets data races (along with atomicity violations). Therefore, we focus on comparing Hippodrome with PFix.

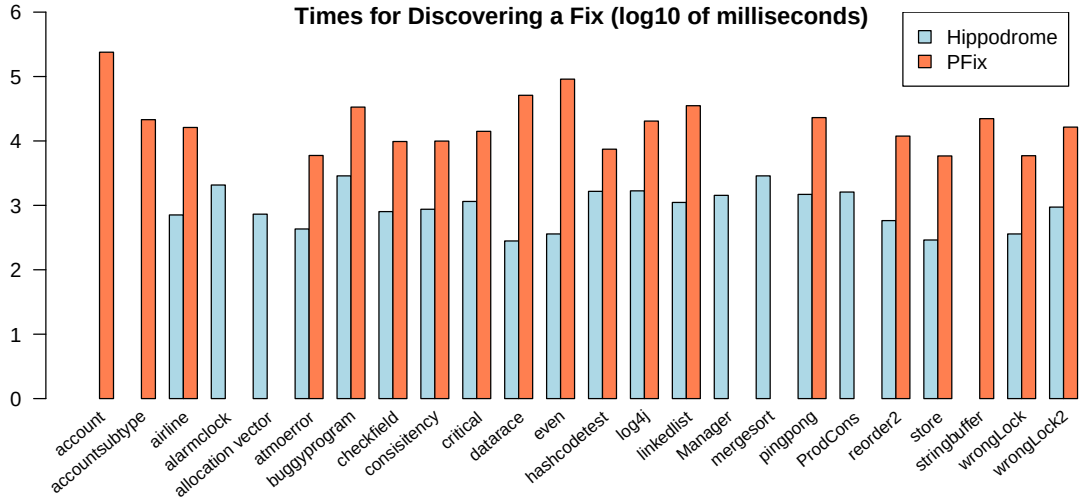
Selection of Subject Applications. We chose 24 unique subject apps from the benchmark suites used by the related works as a baseline for evaluating tools that fix concurrency bugs [1–3, 28]. Our choice of subjects is dictated by the following two aims: (a) to evaluate Hippodrome with regard to various data race patterns and (b) to include all subjects from the PFix suite, thus comparing to it on its home turf.

Tab. 1 contains essential information about the subject apps and provides the evaluation results of PFix and Hippodrome on them. In the following, we discuss the comparison of the two tools *wrt.* the following four aspects:

(a) Bug detection efficiency: how many bugs were found.



(a) Bug detection times



(b) Time to find a fix (logarithmic scale)

Fig. 7. HIPPODROME *vs.* PFIX: run-time comparison.

(b) Performance of both detecting and fixing bugs.
(c) Quality of the produced concurrent patches.
(d) Fundamental limitations of both approaches.

Comparing bug detection efficiency. HIPPODROME leverages the enhanced version of RACERD (*cf.* Sec. 3) for bug detection and creates repairs for *all* detected bugs in the selected subject apps. The seventh column in Tab. 1 shows the detection rate on the subject programs. In particular, HIPPODROME failed to fix three programs, all due to missed bugs. A close inspection revealed that all bug misses correspond to atomicity violations inside an already synchronised block,

which is consistent with the goal of only detecting races. RacerD detected one of the two atomicity violations [46] in `buggyprogram` (manifested as a race); thus, Hippodrome only fixed the detected violation—we mark this as a partial fix.

To fix a data race, PFix requires memory access patterns of passing and failing test executions obtained by running Java PathFinder (JPF) [65] and Unicorn [55] (both 100 times). The patterns are ranked based on their occurrence in the test executions. Each run of PFix generates a repair to be validated using 100 test runs. If the bug is still observed, PFix reruns continuing the process until a failure is not observed for all 100 test runs. PFix failed to fix *five programs* in our selection; in two cases, JPF failed to obtain a failing test execution, and in three instances, PFix was unable to create a successful fix.

Comparing runtime performance. Fig. 7 compares the run-time performance of Hippodrome to PFix. Fig. 7a provides the detection time and Fig. 7b provides the time taken to fix. The total time (detection and fix) in PFix does *not* include the time to validate a fix, whereas the total time of Hippodrome also includes validation. Hippodrome first detects all the bugs in a program, a process which includes both the compilation and the analysis. In 13 instances, RacerD was invoked twice (last column in Tab. 1), i.e., once to detect the bugs and another to validate the fix. In other cases, it was invoked multiple times (within a single run of Hippodrome) to validate different repair options. For example, in the `critical` program, Hippodrome inferred five patches, thus, RacerD ran six times and took 5.8 seconds. Particularly, a repair is discarded if the validation run of RacerD detects a deadlock. As per Fig. 7b, Hippodrome outperforms PFix *by several orders of magnitude.*

Comparing patch quality. In 14 out of 24 cases, Hippodrome produced repairs equivalent to PFix's patches. In four cases, Hippodrome failed to detect bugs or succeeded partially, while in six cases, it created better fixes than PFix: the latter either missed those bugs or created an incomplete fix. For example, Hippodrome's repair fixes the data races in both the `withdrawal` and `deposit` methods in Fig. 2 by wrapping them into `synchronized` blocks on the same lock. The patch produced by PFix ignores the data race in `withdrawal` because its input memory access patterns do not contain the bug path for that method. This example highlights the basic problem with tools relying on dynamic approaches: identifying a subset of buggy paths might be insufficient for producing a complete fix.

Inherent downsides of PFix and Hippodrome. Apart from the inherited inefficiencies from JPF (e.g., scalability to large Java programs), PFix's methodology suffers from randomness in the ranking of failure-inducing memory access patterns. This results in several incorrect repairs involving thus multiple validations and degrading efficiency. Our experiments with simple test cases (e.g., Appendix 8) demonstrate major flaws in PFix's repair process. Besides, PFix chooses a locking policy dynamically by monitoring the lock acquisition patterns during the shared variable access. The incomplete nature of this policy makes the suggested repairs deadlock-prone (confirmed in [38, §4.4]). On the other hand, Hippodrome's expressivity is limited by the power of the underlying analysis, which makes it applicable exclusively to repairing data races, but not, e.g., atomicity violations.

We conclude this case study by responding to **RQ1**:

> *Hippodrome generates high-quality repairs for 20 out of 24 reported bugs in a benchmark suite of the state-of-the-art tool PFix. In all successful cases, Hippodrome significantly outperforms PFix in terms of runtime, while its patch quality is the same or better in the majority of the cases.*

Hippodrome results on JaConTeBe subjects. We next measured the rate of producing data race fixes in real-world Java projects selected from the JaConTeBe benchmark [39]. The JaConTeBe benchmark consists of 47 Java test classes

Table 2. Detection and fix rate for a subset of JaConTeBe. Average results times are given in seconds

Program ID	RACERD	HIPPODROME		
	# Det./# Conf.	# Fixed/# Det.	Avg. Det. Time	Avg. Fix Time
dbcp	0/1	0/0	N/A	N/A
derby	1/1	1/1	1.69	2.87
groovy	1/5	1/1	1.35	5.39
jdk	3/10	3/3	1.47	1.8
log4j	2/2	2/2	1.45	2.07
pool1	1/1	1/1	1.32	3.6

Table 3. Number of bugs detected by RACERD on large scale projects

Project Name	# Confirmed Bugs	# Detected by RACERD
Apache Tomcat	43	35
Google Guava	7	4

that demonstrate concurrency faults in both standard Java library classes and in Java libraries obtained from third parties. We solely focus on the subjects which contain various data race patterns. We summarise our findings in Tab. 2, highlighting: the detection rate (second column: the number of data races detected by RACERD versus the number of officially confirmed ones), the fix rate (third column: how many of the detected data races has HIPPODROME fixed), and the average time to report and to fix a bug (column four and five, respectively). A more detailed version of this experiment indicating the exact version of each selected library can be found in the Appendix, Tab. 6.

RACERD discovered eight of the twenty data races confirmed to manifest in JaConTeBe. Seven of the missed data races involve shared resource accesses from within exception code blocks—RACERD can not identify such races because its intermediate representation language, i.e. SMALLFOOT, does not have support for exception handling reasoning. The remaining unsuccessful cases include threads spawned from the main method (violates assumption (b) stated in Sec. 4.4) and code from non-included libraries (for example, in the jdk6_2, mkdir bug). All of the detected bugs were successfully repaired by HIPPODROME. PFIX could not execute on any of the twenty subjects, owing mainly to JPF's inability to handle these subjects (known limitation on JaConTeBe subjects).

5.4 RQ2: HIPPODROME and Human Fixes

We evaluated HIPPODROME's performance, correctness, and quality of its patches on known concurrency bugs in two large-scale open-source Java projects, namely Apache Tomcat and Google Guava. We first searched for the occurrences of the keywords *concur*, *thread*, *sync*, *lock*, and *race* in the commit history of Apache Tomcat and Google Guava over the past five years. Next, we ran RACERD on each version preceding the commit that fixed a bug and manually checked whether the fixed bug was detected.[5] In particular, we analysed 50 concurrency bugs and RACERD successfully reported 39 of those, findings summarised in Tab. 3.

Statistics of developers' fixes. Tab. 4 summarises different kinds of fixes introduced by developers. In particular, 69 code changes were performed to repair 50 races. Making a shared variable `volatile` was the most popular approach

[5]Few cases required a single `@ThreadSafe` annotation to the buggy class.

Table 4. Developers' fix strategy

Project	Added Volatile	Added sync block	Changed Collection	Others
Tomcat	32	14	7	7
Guava	3	4	2	0

(35 out of 69) to data race repair, although they could have also been fixed via `synchronized`. At the same time, the developer's input is also needed to validate the addition of `volatile`, as affected writes may become expensive. Because of this, Hippodrome suggests adding `volatile` as a secondary fix option, favouring the use of `synchronized` as a primary fix. In few cases, developers changed the type of Java collections to their thread-safe variants, e.g., `HashMap` to `ConcurrentHashMap`. This kind of fix is beyond the scope of this work. The column 'Others' includes repairs such as replacing the lock object or the type of lock block and making the shared variable *final*(#066e25467). For example, in one case, a *synchronized* block was replaced with the *readLock.lock()* (#be19e9b1e). The use of *readLock* may improve the performance by allowing read access to multiple threads, while no threads hold the *writeLock*. This performance benefit is realized when reads are more frequent. We do not consider *readock*/*writeLock*, as counting the frequency of reads and write is out of scope for this work.

Repair quality. Tab. 5 provides Hippodrome's results on 39 developers' commit. Column *Commit#* describes the commit under which the developer fixed a particular concurrency bug. *Fix status* denotes whether Hippodrome successfully fixed a bug, and *Detection/Fix Time* shows the time taken to detect/fix a bug. Finally, *equivalence* shows the syntactic and semantic equivalence of the repair. A patch is semantically equivalent to the developer's patch if it fixes the specific bug similarly to the developer's patch but does not use the same syntax. For example, both `volatile` and `synchronized` constructs may be used to repair a race and thus provide semantically equivalent results. RacerD failed to detect 8 bugs in Tomcat (out of 43) and 3 in Guava (out of 7); thus, we ran Hippodrome on 39 remaining bugs. Hippodrome generated 83% correct fixes for Tomcat (29/35), and 50% for Guava (2/4). In particular, it failed to repair 3 bugs requiring to add interface-level annotations which are beyond the scope of this work. In the remaining 5 fixes, developers introduced new features, thus we cannot draw a direct comparison to those.

Fig. 8 shows Hippodrome's performance on the selected 39 bugs. Detection and validation took about one minute, most of which is attributed to compilation.[6] RacerD detection time is usually shared to detect multiple bugs, yet in this experiment we ran it from a "cold" setup, zooming in on each particular bug. The AST creation for project files takes most of the time to find a fix; the actual fix generation takes less than a second. Therefore, amortised times for detecting and fixing several bugs in a batch mode would be much smaller.

Repairs for 18 bugs were syntactically the same as the developers' fixes. In 2 cases Hippodrome generated better repairs, introducing *less* overhead. In one instance, the developer's repair was better: our repair locked multiple `case` clauses in a `switch` statement, instead of just the race-causing clause. Besides analyzing the RacerD's signal, the equivalence was manually validated by at least three authors in our study.

An attempt to run PFix on these large-scale projects produced no fix report irrespectively of the set timeout.[7]

Hippodrome repairs 31 out of 39 concurrency bugs. 60% of these fixes are syntactically the same as the developer patches. Hippodrome runtime performance scales well for large-projects with about 80 seconds to fix the first bug.

[6] In its standard mode, RacerD attaches itself to the compilation process.
[7] This is due to inherent scalability limitations of JPF, which we explicitly confirmed in our communication with the PFix authors.

Table 5. HIPPODROME results on Developers' commits. All times are given in seconds.

Index	Tomcat	HIPPODROME			
	Commit#	Fix Status	Detection Time	Fix Time	Equivalence
1	752f0b9f	Fail	NA	NA	NA
2	582cc729	Success	87.92	21.47	Syntactic
3	5e9f6fd6	Success	88.25	15.95	Syntactic
4	066e2546	Fail	NA	NA	NA
5	317480b9	Success	82.29	17.85	Semantic
6	7040497f	Success	76.84	19.32	Semantic
7	29f060ad	Fail	NA	NA	NA
8	b96f9bec	Success	77.94	30.75	Semantic
9	0ca05961	Success	14.75	6.6	Syntactic
10	be19e9b1	Fail	NA	NA	NA
11	4caec93b	Fail	NA	NA	NA
12	ed610381	Success	73.84	23.28	Syntactic
13	057de944	Success	71.09	20.6	Semantic
14	518c27c3	Success	72.3	14.65	Semantic
15	227b6093	Success	75.09	7.84	Syntactic
16	fb631d21	Success	74.63	6.34	Semantic
17	50121380	Success	80.46	17.57	Semantic
18	d85c35f	Success	77.29	54.6	Semantic
19	3360c3a	Success	56.28	42.3	Semantic
20	8cbb9f8	Success	112.61	10.37	Semantic
21	afff25f1c	Success	55.84	15.18	Syntactic
22	d4c8da6	Success	55.28	20.67	Syntactic
23	d9b530c	Success	55.19	19.15	Syntactic
24	e80797f3	Success	55.34	18.67	Syntactic
25	1484f3ec	Success	57.49	7.55	Semantic
26	825c450c	Success	55.27	9.38	Semantic
27	02a4bb92	Success	56.99	4.13	Semantic
28	3d6dbd91	Success	55.36	15.36	Syntactic
29	8313fa0f1	Success	52.57	17.4	Syntactic
30	52b29fd2	Success	54.82	21.3	Syntactic
31	ea383db5	Success	55.39	11.46	Syntactic
32	cadbc500	Fail	NA	NA	NA
33	5379ae68	Success	55.4	19.67	Syntactic
34	3078444	Success	52.27	9.86	Syntactic
35	17b6c64f	Success	55.17	15.48	Syntactic
	Guava				
36	0e94fb5b	Success	24.74	10.033	syntactic
37	0e94fb5b	Success	24.67	80.02	syntactic
38	c15cd804	Fail	NA	NA	NA
39	a43b4aa7	Fail	NA	NA	NA

5.5 Example of the Impact of Bug Clustering on Patch Quality

Although the bug reports generated by RACERD involve at most two insufficiently synchronised accesses to a shared resource, our bug clustering mechanisms recovers the holistic view of accesses to the shared resource, thus leading to patches a human is more likely to favour. Fig. 9 presents a simplified data race example to demonstrate the benefits of bug clustering (please see Fig. 10 and Fig. 11 in the Appendix for full code details). This program creates three threads (referred to as `t1`, `t2`, and `t3` for simplicity) with separate entry points at Line 4, Line 20, and Line 33, respectively. There is a data race among `t1` and `t2` due to unprotected accesses to global variable `sharedString` at Line 11 and 26,

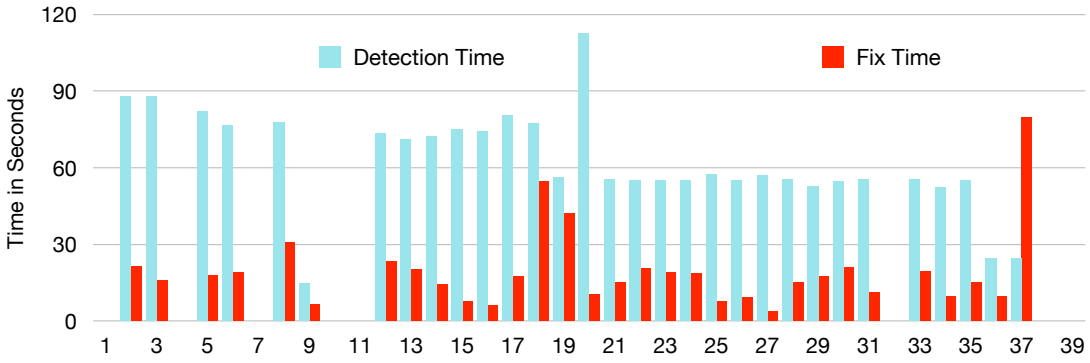


Fig. 8. HIPPODROME run-time on large-scale projects

respectively. `t1` protects the access to `sharedString` via a lock `l`, while `t2` uses the lock `p`. At the same time, `t3` protects the access to `sharedString` with three locks in the following order: `GlobalLock`, `p`, `l`.

A correct patch should choose a lock object such that the resulting program cannot lead to a race on `sharedString` anymore (on top of making sure that it produces no deadlock with already existing ones). However, as exemplified by Fig. 9a which depicts in comments a couple of the few repairs produced by PFIX, automatically producing such patches is not always straightforward when each bug is treated as an isolated candidate for repair. We ran PFIX ten times and none of the generated fixes were correct. In contrast, our patch synthesis strategy generates multiple correct patches—see two of the produced patches in comments in Fig. 9b. One patch simply prevents the data race (`Fix-1`), and another more natural patch (`Fix-2`) takes a synchronising format that seems more similar to what a programmer might write on top of fixing the data race correctly.

6 THREATS TO VALIDITY

This section discusses the threats to the internal and external validity of our experiments and their mitigation solutions.

Internal Validity. Our approach relies on existing static analysis tools for bug detection. In particular, HIPPODROME employs RACERD for data race detection, hence our experiments are sensitive to the efficiency and correctness of RACERD. Sec. 4.4 details the analysis's correctness criteria and the assumptions that need to hold for it to detect all data races. When known bugs were not detected in our experiments, we manually investigated the underlying reasons and reported the assumptions which were violated. Future work could investigate how to check whether these assumptions hold at runtime, e.g. via dynamic monitoring. Furthermore, our approach relies on the availability of the complete source code pertaining to a data race bug. To mitigate this constraint, library calls whose source code is unavailable can be ascribed access summaries manually, or modelled accordingly within the detection tool, e.g. we instructed RACERD with models for the `read` and `write` methods of `java.net.ServerSocket` for one of the data races in Apache Tomcat.

HIPPODROME also relies on static analysis tools to check how the synthesised patch interacts with the existing locks, i.e. whether it can lead to a deadlock. We employed RACERD for this phase, too, since the tool is equipped with deadlock detection mechanisms. Again, the assumptions under which no deadlock is missed are summarized in Sec. 4.4. We have found no example in our extensive testing and during the evaluation where HIPPODROME introduces buggy patches, except for testing subjects that violate the correctness assumptions, e.g. subjects manipulating anonymous classes.

```java
class HelloHelper {
  private String sharedString = null;
// static Object objectFix = new Object(); -- Fix-2 (1/3)
  public void t1() {

      synchronized(l) {
        // synchronized (this){           -- Fix-1 (1/4)
         System.out.println("T1 got lock l");
        //}                               -- Fix-1 (2/4)
         sharedString = "Hello World-1 ";
         sharedString = sharedString + "from T1";
         System.out.println(sharedString);
         // synchronized (this){          -- Fix-1 (3/4)
         // synchronized (objectFix){     -- Fix-2 (2/3)
         System.out.println("T1 released lock l");
         //}                              -- Fix-1 (4/4)
         //}                              -- Fix-2 (3/3)
   }   }

  public void t2() {
    synchronized(p) {

       sharedString = "Hello World-2 ";


       sharedString = sharedString + "from T2";


       System.out.println(sharedString);

    }  }

  public void t3() {
    synchronized(GlobalLock) {
      synchronized(p) {
        synchronized(l) {
          sharedString = "Hello World-3 ";
          sharedString = sharedString + "from T3";
          System.out.println(sharedString);
        }
      }
  } } }
```

(a) PFIX Repairs

```java
class HelloHelper {
  private String sharedString = null;

  public void t1() {
    // synchronized(p) {              -- Fix-2 (1/2)
    synchronized(l) {

      System.out.println("T1 got lock l");

      sharedString = "Hello World-1 ";
      sharedString = sharedString + "from T1";
      System.out.println(sharedString);


      System.out.println("T1 released lock l");
    }
    // }                              -- Fix-2 (2/2)
  }

  public void t2() {
    synchronized(p) {
      // synchornized(l) {           -- Fix-1 (1/6)
      sharedString = "Hello World-2 ";
      // }                           -- Fix-1 (2/6)
      // synchornized(l) {           -- Fix-1 (3/6)
      sharedString = sharedString + "from T2";
      // }                           -- Fix-1 (4/6)
      // synchornized(l) {           -- Fix-1 (5/6)
      System.out.println(sharedString);
      // }                          -- Fix-1 (6/6)
  }  }

  public void t3() {
    synchronized(GlobalLock) {
      synchronized(p) {
        synchronized(l) {
          sharedString = "Hello World-3 ";
          sharedString = sharedString + "from T3";
          System.out.println(sharedString);
        }
      }
  } } }
```

(b) HIPPODROME Repairs

Fig. 9. A data race example in Java

One final concern is that of the selection of the evaluation subjects, i.e., whether the chosen subjects favour HIPPODROME. We mitigate this threat by focusing on subjects commonly found in related works. We evaluated HIPPODROME on various data race patterns from benchmarks that serve as a baseline to assess tools for repairing concurrency bugs [1–3], including those used by PFIX.

External Validity. Threats to external validity concern the generalizability of our outcomes, i.e. our findings may not extend outside the chosen micro-benchmarks. To mitigate this threat, the subjects in our second study have been drawn from 50 known real-world concurrency bugs in two large-scale Java projects, namely Apache Tomcat and Google Guava. We chose Apache Tomcat since it is noted for concurrency issues and has been studied in earlier efforts [42].

We chose Google Guava due to its widespread usage in industry, and because it offers a variety of concurrency features. To consider the latest developments and scalability, we incorporated concurrency bugs from these projects' commit histories over the last five years. Hippodrome performed well on these subjects.

7 RELATED WORK

General Program Repair and Synthesis. Automated program repair (APR) [25] is an emerging technology paradigm for automatically fixing logical bugs via search [67], semantic reasoning [53] and learning [45]. The recent works on *semantic* program repair [49, 53] make use of advances in program synthesis [16, 48, 54, 56] to automatically generate one-line or multi-line fixes. However, these approaches have been mostly studied for sequential programs.

Program Repair for Concurrency. While PFix [38] is a recent work in this direction, there are other efforts on concurrency repair, such as [12, 29, 32, 33, 36–38, 41–43] to name a few. Many of these tools use dynamic analysis to find bugs, while some statically validate fixes [32, 33]. In general, dynamic analysis approaches may miss concurrency bugs, and the repairs may be incomplete. PFix [38] denotes a somewhat hybrid approach which relies on the JPF model checker for bug finding. Apart from the scalability limitations of JPF, such an approach requires providing the temporal properties to the model checker—which PFix provides by exploiting an incomplete set of likely failure inducing memory access patterns. AlphaFixer [12] is an approach to APR for atomicity violations for C++ programs which analyses the lock acquisitions in order to reduce the introduction of deadlocks. Close examination revealed that its underlying repair algorithm is unsound in that it introduces deadlocks in basic examples where atomicity violations manifest as data races. We refer the reader to Fig. 12 for the C++ counterpart of the example discussed in Sec. 5.5, and a fix which introduces a deadlock produced by AlphaFixer[8]. Fig. 11 shows the corresponding Hippodrome deadlock-free fix. The work on HFix [41] falls into the category of using syntactic or pattern matching-based static analysis, where patterns of patches are obtained by mining human patches. Such an approach is inherently limited to the data set of human patches considered, and carry no guarantees of correctness. Grail, also based on static analysis [42], offers patches which are guaranteed to be deadlock-free, although these guarantees have been shown by [38] to not always hold in practice. Unlike our work, Grail is not modular; it relies on mixed-integer programming computation and Petri net models which restrict the fixes to single classes or methods. Bugs spanning multiple classes or methods pose no problem for Hippodrome as shown by our experiments.

Program Repair with Static Analysis. Logozzo and Ball co-authored an early work using abstract interpretation based static analysis for sequential program repair [44]. Unlike our work which requires no user annotations, their approach needs formal safety properties to drive the repair. The work on AFix focuses on atomicity violations [32]. AFix and its later incarnation CFix [33] represent pattern-based approaches for fixing concurrency bugs; these techniques do not come with correctness guarantees. AFix is in fact known to introduce deadlocks as pointed out in [42]. Similarly, the work on Phoenix [8] also patches static analysis warnings via repair strategy examples learnt from big codebases. The recent work on FootPatch is closer to our theme of analysis-driven correct repairs [64]. FootPatch exploits the *locality* in the summaries inferred by a compositional static analysis [13] for efficiently generating sound one-line patches for imperative programs, fixing memory leaks and null-pointer exceptions. Due to a non-local nature of concurrency issues (e.g., data races), FootPatch cannot be used directly for fixing concurrency errors, which is achieved in this paper.

[8]We obtained that fix by running the latest version of AlphaFixer which we obtained directly from its authors.

Static Analysis for Concurrency Bugs. Statically detecting concurrency bugs is a well researched, yet notoriously difficult problem. Exhaustive exploration is often infeasible even for medium-sized programs, yielding imprecise results [21]. To improve precision, effort has been made in adding the user in the process of specifying what properties the code of interest should have [15, 31, 50, 61]. Such efforts provide high accuracy, but their friction is too high for their adoption in APR. RACERX [18], an influential automatic static data race detector, also works on building summaries by approximating the locks held at each program point. However, it has low signal as shown at the experiments done at Meta [10]. CHORD, a more advanced approach with high recall, is shown to be too imprecise and too slow for CI integration [51]. A number of commercial automated tools exists too [14, 60, 62]. We opted for an open-source one in order to identify what analysis components can APR leveraged on, e.g. the access summaries. DEPCON is a static analysis building on similar observations as RACERD, namely that only methods that *may* both interleave and access the same shared memory locations can lead to data races or atomicity violation [63]. While this tool is tuned for concurrency tests generation, it would be interesting to investigate its feasibility as a support for program repair in the future.

Works exists also for the static detection of atomicity violations [22, 59, 66], however, since the current goal of HIPPODROME is to fix data races, we leave the investigation of fixing atomicity violations as a future work. Although numerous works exists for the detection of deadlocks in Java programs [6, 52, 68], our validation phase relies on the analysis which comes in the same package with RACERD for convenience purposes, but also because it is accompanied by a formal proof of correctness.

8 CONCLUSION

We have described a static analysis-driven automated program repair technique for concurrent programs which is scalable, modular, and preserves deadlock-freedom. It can be fitted into a Continuous Integration loop, which can allow collaboration with developers to gradually improve automatically generated patches. We have shown through experiments that a synergy between static analysis bug detection and data race repair yields good quality patches even for large scale code bases, such as Tomcat and Guava. Furthermore, our work also highlights what features and components a static analysis should have, for it to support the patch synthesis process—an information which could guide the design of future static analysis tools with repair in mind. For example, the information captured in the access summaries is useful in deciding the clustering criteria, or in choosing candidates for the lock object.

Our work can be used to further explore related research avenues: richer solution spaces with more synchronization primitives, application-dependent cost functions for patches, lock choice heuristics. In particular, staging the patch synthesis processes into Intermediate Representations makes it convenient for future work to extend the lock support to finer-grained synchronization primitives. Clustering the bugs according to the shared resource allowed us to experiment with a modest solution for fixing atomicity violations even if the bug detection analysis was only designed for reporting data races. We believe this opens further research opportunities, by considering more complex atomicity violation. The bug clustering mechanisms also led to better quality patches by recovering the holistic view on the origins of the data races while retaining compositionality. This approach produces better quality patches, however we believe there is space to further explore the granularity and criteria of this clustering so as to design smarter lock choice heuristics and cost functions. For example, it would be interesting to distinguish between the cases where intrinsic locks would be the ideal candidate, from those where intrinsic locks would reduce concurrency.

Since our publicly available tool, HIPPODROME, can also be used in interactive mode, it can be further enhanced to create different user studies in order to understand solution comprehension and preference. For example, we could

create a study to understand whether developers would favour a simple fix which is not syntactically invasive to one which involves more code changes but results in better overall concurrency.

ACKNOWLEDGEMENTS

This work was partially supported by the Singapore MoE Tier 3 grant MOE-MOET32021-0001, by the National Research Foundation Singapore (National Satellite of Excellence in Trustworthy Software Systems) and by Singapore MoE Tier 1 Grant No. IG18-SG102.

REFERENCES

[1] 2011. The Pecan Benchmarks. https://www.cse.ust.hk/prism/pecan/#Experiment
[2] 2016. JaConTeBe Object Biography. http://sir.csc.ncsu.edu/portal/bios/concurrency.php
[3] 2018. The PFix Benchmarks. https://github.com/PFixConcurrency/FixExamples
[4] 2021. Hippodrome (Docker Image). https://hub.docker.com/u/hippodrome
[5] 2021. Hippodrome (Main Repository). https://github.com/verse-lab/hippodrome
[6] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. 2006. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *HVC*. Springer Berlin Heidelberg.
[7] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Softw.* 25, 5 (2008), 22–29.
[8] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: automated data-driven synthesis of repairs for static analysis violations. In *ESEC / SIGSOFT FSE*. ACM, 613–624.
[9] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.
[10] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 144:1–144:28.
[11] James Brotherston, Paul Brunet, Nikos Gorogiannis, and Max I. Kanovich. 2021. A Compositional Deadlock Detector for Android Java. In *ASE*. 955–966. https://doi.org/10.1109/ASE51524.2021.9678572
[12] Yan Cai, Lingwei Cao, and Jing Zhao. 2017. Adaptively Generating High Quality Fixes for Atomicity Violations. In *ESEC/FSE 2017*. ACM, 303–314.
[13] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *POPL*. ACM, 289–300.
[14] Andy Chou. 2014. From the Trenches: Static Analysis in Industry. Invited keynote talk at POPL'14. Available at https://popl.mpi-sws.org/2014/andy.pdf.
[15] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, 23–42.
[16] Andreea Costea, Amy Zhu, Nadia Polikarpova, and Ilya Sergey. 2020. Concise Read-Only Specifications for Better Synthesis of Programs with Pointers. In *ESOP (LNCS, Vol. 12075)*. Springer, 141–168.
[17] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.
[18] Dawson R. Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP*. ACM, 237–252.
[19] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *PLDI*. ACM, 121–133.
[20] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *PLDI*. ACM, 110–121.
[21] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*. ACM, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315
[22] Damian Giebas and Rafał Wojszczyk. 2020. Atomicity Violation in Multithreaded Applications and Its Detection in Static Code Analysis Process. *Applied Sciences* 10, 22 (2020). https://doi.org/10.3390/app10228005
[23] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice*. Addison-Wesley.
[24] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A True Positives Theorem for a Static Race Detector. *Proc. ACM Program. Lang.* 3, POPL (2019), 57:1–57:29.
[25] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.
[26] Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. M. Kaufmann.
[27] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. In *PLDI*. ACM, 337–348.

[28] Jeff Huang and Charles Zhang. 2011. Persuasive Prediction of Concurrency Access Anomalies. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 144–154. https://doi.org/10.1145/2001420.2001438

[29] Jeff Huang and Charles Zhang. 2012. Execution privatization for scheduler-oblivious concurrent programs. In *OOPSLA*. ACM, 737–752.

[30] DeLesley Hutchins, Aaron Ballman, and Dean Sutherland. 2014. C/C++ Thread Safety Analysis. In *SCAM*. IEEE, 41–46.

[31] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NFM*. 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

[32] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *PLDI*. ACM, 389–400.

[33] Guoliang Jin, Wei Zhang, and Dongdong Deng. 2012. Automated Concurrency-Bug Fixing. In *OSDI*. USENIX Association, 221–236.

[34] Neil D Jones and Steven S Muchnick. 1979. Flow analysis and optimization of LISP-like structures. In *POPL*. Association for Computing Machinery, 244–256.

[35] Sepideh Khoshnood, Markus Kusano, and Chao Wang. 2015. ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs. In *ISSTA*. ACM, 165–176.

[36] Bohuslav Krena, Zdenek Letko, Rachel Tzoref, Shmuel Ur, and Tomáš Vojnar. 2007. Healing Data Races On-the-Fly. In *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*. ACM, 54–64.

[37] Bohuslav Křena, Zdeněk Letko, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, Shmuel Ur, and Tomáš Vojnar. 2009. *A Concurrency Testing Tool and Its Plug-Ins for Dynamic Analysis and Runtime Healing*. Springer-Verlag, 101–114.

[38] Huarui Lin, Zan Wang, Shuang Liu, Jun Sun, Dongdi Zhang, and Guangning Wei. 2018. PFix: fixing concurrency bugs based on memory access patterns. In *ASE*. ACM, 589–600.

[39] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 178–189. https://doi.org/10.1109/ASE.2015.87

[40] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. When Threads Meet Events: Efficient and Precise Static Race Detection with Origins. In *PLDI*. ACM, 725–739.

[41] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and generating high quality patches for concurrency bugs. In *FSE*. ACM, 715–726.

[42] Peng Liu, Omer Tripp, and Charles Zhang. 2014. Grail: context-aware fixing of concurrency bugs. In *FSE*. ACM, 318–329.

[43] Peng Liu and Charles Zhang. 2012. Axis: Automatically fixing atomicity violations through solving control constraints. In *ICSE*. IEEE Computer Society, 299–309.

[44] Francesco Logozzo and Thomas Ball. 2012. Modular and verified automatic program repair. In *OOPSLA*. ACM, 133–146.

[45] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL*. ACM, 298–312.

[46] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*. ACM, 329–339.

[47] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. 2008. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*. IEEE Computer Society, 277–288.

[48] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic execution with existential second-order constraints. In *ESEC / SIGSOFT FSE*. ACM, 389–399.

[49] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *ICSE*. 691–701.

[50] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, 41–62.

[51] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *PLDI*. ACM, 308–319.

[52] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective static deadlock detection. In *ICSE*. IEEE, 386–396.

[53] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *ICSE*. IEEE Computer Society, 772–781.

[54] Thanh-Toan Nguyen, Quang-Trung Ta, Ilya Sergey, and Wei-Ngan Chin. 2021. Automated Repair of Heap-Manipulating Programs Using Deductive Synthesis. In *VMCAI (LNCS, Vol. 12597)*. Springer, 376–400.

[55] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. 2012. A Unified Approach for Localizing Non-deadlock Concurrency Bugs. In *ICST*. IEEE Computer Society, 51–60.

[56] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 72:1–72:30.

[57] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66.

[58] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. 2015. Synthesizing racy tests. In *PLDI*. ACM, 175–185.

[59] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. 2005. Automated Type-Based Analysis of Data Races and Atomicity. In *PPoPP*. ACM, 83–94. https://doi.org/10.1145/1065944.1065956

[60] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. *Proceedings of the Workshop on Binary Instrumentation and Applications*, 62–71.

[61] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-Grained Concurrent Programs. In *PLDI*. ACM, 77–87. https://doi.org/10.1145/2737924.2737964

[62] Fausto Spoto. 2016. The Julia Static Analyzer for Java. In *Static Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 39–57.

[63] Valerio Terragni, Mauro Pezzè, and Francesco Adalberto Bianchi. 2019. Coverage-Driven Test Generation for Thread-Safe Classes via Parallel and Conflict Dependencies. In *ICST*. 264–275. https://doi.org/10.1109/ICST.2019.00034

[64] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *ICSE*. ACM, 151–162.

[65] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. 2004. Test input generation with Java PathFinder. In *ISSTA*. ACM, 97–107.

[66] Christoph Von Praun and Thomas R Gross. 2004. Static Detection of Atomicity Violations in Object-Oriented Programs. *J. Object Technol.* 3, 6 (2004), 103–122.

[67] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE*. IEEE, 364–374.

[68] Amy Williams, William Thies, and Michael Ernst. 2005. Static Deadlock Detection for Java Libraries. In *ECOOP*. 602–629. https://doi.org/10.1007/11531142_26

Table 6. Results on JaConTeBe apps by HIPPODROME. All times are given in seconds. Det. means "detection".

Program ID	Affected **Version**	RACERD **Det.**	HIPPODROME		
			Fix Status	**Det. Time**	**Fix Time**
dbcp3	1,2	No	N/A	N/A	N/A
derby3	10.5.1.1	Yes	Yes	1.69	2.87
groovy1	1.7.9	Yes	Yes	1.35	5.39
groovy3	1.7.9	No	N/A	N/A	N/A
groovy4	1.7.9	No	N/A	N/A	N/A
groovy5	1.7.9	No	N/A	N/A	N/A
groovy6	1.7.9	No	N/A	N/A	N/A
jdk6_1	1.6.0	Yes	Yes	1.25	0.81
jdk6_2	[1.6.0 - 1.6.0_02)	No	N/A	N/A	N/A
jdk6_3	1.6.0	No	N/A	N/A	N/A
jdk6_4	1.6.0	No	N/A	N/A	N/A
jdk6_5	1.6.0	Yes	Yes	2.01	2.48
jdk6_13	1.6.0	No	N/A	N/A	N/A
jdk6_14	1.6.0	No	N/A	N/A	N/A
jdk7_1	1.7.0	Yes	Yes	1.14	2.1
jdk7_3	[1.7.0 - 1.7.0_40)	No	N/A	N/A	N/A
jdk7_6	1.7.0	No	No	N/A	N/A
log4j1	1.2.15	Yes	Yes	1.07	2.16
log4j3	1.2.13	Yes	Yes	1.83	1.98
pool1	1,4	Yes	Yes	1.32	3.6

APPENDIX

Fig. 10 and Fig. 11 are the extended code versions of Fig. 9a and Fig. 9b, respectively. Tab. 6 is the extended version of Tab. 2.

Fig. 10. A data race example in Java (PFix Repairs)

```
1 class HelloHelper {
2   private String sharedString = null;
3 // static Object objectFix = new Object();    -- PFix Fix-2 (1/3)
4   public void t1() {
5       synchronized(l) {
6        // synchronized (this){               -- PFix Fix-1 (1/4)
7         System.out.println("T1 got lock l");
8       //}                                    -- PFix Fix-1 (2/4)
9         sharedString = "Hello World-1 ";
10        sharedString = sharedString + "from T1";
11        System.out.println(sharedString);
12        // synchronized (this){             -- PFix Fix-1 (3/4)
13      //   synchronized (objectFix){        -- PFix Fix-2 (2/3)
14        System.out.println("T1 released lock l");
15      //}                                    -- PFix Fix-1 (4/4)
16       //}                                   -- PFix Fix-2 (3/3)
17      }
18  }
19  public void t2() {
20    synchronized(p) {
21      System.out.println("T2 got lock p");
22
23       sharedString = "Hello World-2 ";
24       sharedString = sharedString + "from T2";
25       System.out.println(sharedString);
26
27      System.out.println("T2 released lock p");
28    }  }
29  public void t3() {
30    synchronized(GlobalLock) {
31      System.out.println("T3 got lock G");
32      synchronized(p) {
33        System.out.println("T3 got lock p");
34        synchronized(l) {
35          System.out.println("T3 got lock l");
36
37          sharedString = "Hello World-3 ";
38
39          sharedString = sharedString + "from T3";
40          System.out.println(sharedString);
41          System.out.println("T3 released lock l");
42        }
43        System.out.println("T3 released lock p");
44      }
45      System.out.println("T3 released lock G");
46    } } }
47 public class Hello {
48   public static void main(String[] args){
49     HelloHelper h = new HelloHelper();
50     Thread thread1 = new Thread(){ public void run() { h.t1(); } };
51     Thread thread2 = new Thread(){ public void run() { h.t2(); } };
52     Thread thread3 = new Thread(){ public void run() { h.t3(); } };
53
54     thread1.start();
55     thread2.start();
56     thread3.start();
57   } }
```

Fig. 11. A data race example in Java (HIPPODROME Repairs)

```
class HelloHelper {
  private String sharedString = null;

  public void t1() {
    // synchronized(p) {                          -- Hippodrome Fix-2 (1/2)
      synchronized(l) {
        System.out.println("T1 got lock l");
        sharedString = "Hello World-1 ";
        sharedString = sharedString + "from T1";
        System.out.println(sharedString);
        System.out.println("T1 released lock l"); }
      // }                                         -- Hippodrome Fix-2 (2/2)
  }
  public void t2() {
    synchronized(p) {
      System.out.println("T2 got lock p");
      // synchornized(l) {                        -- Hippodrome Fix-1 (1/6)
        sharedString = "Hello World-2 ";
        // }                                       -- Hippodrome Fix-1 (2/6)
      // synchornized(l) {                        -- Hippodrome Fix-1 (3/6)
        sharedString = sharedString + "from T2";
        // }                                       -- Hippodrome Fix-1 (4/6)
      // synchornized(l) {                        -- Hippodrome Fix-1 (5/6)
        System.out.println(sharedString);
        // }                                       -- Hippodrome Fix-1 (6/6)
      System.out.println("T2 released lock p");
  }  }
  public void t3() {
    synchronized(GlobalLock) {
      System.out.println("T3 got lock G");
      synchronized(p) {
        System.out.println("T3 got lock p");
        synchronized(l) {
          System.out.println("T3 got lock l");

          sharedString = "Hello World-3 ";

          sharedString = sharedString + "from T3";
          System.out.println(sharedString);
          System.out.println("T3 released lock l");
        }
        System.out.println("T3 released lock p");
      }
      System.out.println("T3 released lock G");
} } }
public class Hello {
  public static void main(String[] args){
    HelloHelper h = new HelloHelper();
    Thread thread1 = new Thread(){ public void run() { h.t1(); } };
    Thread thread2 = new Thread(){ public void run() { h.t2(); } };
    Thread thread3 = new Thread(){ public void run() { h.t3(); } };

    thread1.start();
    thread2.start();
    thread3.start();
} }
```

Fig. 12. A data race example in C++ (ALPHAFIXER Repair)

```
1   struct thread_data {
2     char ⋆message;
3   }
4   struct thread_data ⋆my_data = new thread_data;
5   //...
6   void threadA(void arg) {
7     pthread_mutex_lock(&l);
8     cout << "T1 got lock l\n";
9     // pthread_mutex_lock(& my->GateLock);     -- AlphaFixer Fix (1/4)
10    my_data->message = "Hello World from T1\n";
11    // pthread_mutex_unlock(& my->GateLock);   -- AlphaFixer Fix (2/4)
12    cout << "T1 released lock l\n";
13    pthread_mutex_unlock(&l);
14    pthread_exit(NULL);
15  }
16
17  void threadB(void arg) {
18    pthread_mutex_lock(&p);
19    cout << "T2 got lock p\n";
20    // pthread_mutex_lock(& my->GateLock);     -- AlphaFixer Fix (3/4)
21    my_data->message = "Hello World from T2\n";
22    // pthread_mutex_unlock(& my->GateLock);   -- AlphaFixer Fix (4/4)
23    cout << my_data->message;
24    pthread_mutex_unlock(&p);
25    pthread_exit(NULL);
26  }
27
28  void threadC(void arg) {
29    pthread_mutex_lock(&my->GateLock);
30    cout << "T3 got lock G\n";
31    pthread_mutex_lock(&l);
32    cout << "T3 got lock l\n";
33    pthread_mutex_lock(&p);
34    cout << "T3 got lock p\n";
35
36    my_data->message = "Hello World from T3\n";
37
38    cout << "T3 released lock p\n";
39    pthread_mutex_unlock(&p);
40    cout << "T3 released lock l\n";
41    pthread_mutex_unlock(&l);
42    cout << "T3 released lock G\n";
43    pthread_mutex_unlock(&my->GateLock);
44    pthread_exit(NULL);
45  }
46
47  int main () {
48    pthread_t threads[3];
49    //... init code
50    pthread_create(&threads[0], NULL, threadA, NULL);
51    pthread_create(&threads[1], NULL, threadB, NULL);
52    pthread_create(&threads[2], NULL, threadC, NULL);
53    //...
54  }
```